feincms3 Documentation

Release 3.6.1

Feinheit AG

Jul 05, 2022

CONTENTS

Version 3.6.1

feincms3 offers tools and building blocks which make building a CMS that is versatile, powerful and tailor-made at the same time for each project a reachable reality.

It builds on other powerful tools such as Django itself and its excellent standard admin interface, django-content-editor to allow creating and editing structured content and django-tree-queries for querying hierarchical data such as page trees.

The tools can be used for a page CMS, but also work well for other types of content such as news magazines or API backends for mobile apps.

CHAPTER

ONE

FIRST STEPS

Start here if you want to know what feincms3 is and build your first CMS based on feincms3.

1.1 Introduction

1.1.1 Philosophy

feincms3 follows the library-not-framework approach. Inversion of control is avoided as much as possible, and great care is taken to provide useful functionality which can still be easily replaced if anyone wishes to do so.

Replacing functionality should not require using extension points or configuration but simply different glue code, which should be short and obvious enough to be repeated in different projects.

The idea is not necessarily to avoid code, but to avoid all sorts of complexity, whether obvious or not. The cost of abstractions is that there always comes a moment where you have understand the layers beneath, and often the learning curve gets steep quickly.

feincms3 is your Do It Yourself kit for CMS building, as Django is your Do It Yourself kit for website building.

feincms3 only has abstract model classes and mixins. Any concrete classes (for example, the page model) **have** to be added by you in your own project. This is by design, and paves the way for introducing local customizations without having to rely on hooks, extension points and whatnot.

1.1.2 Standing on the shoulders

Django's builtin admin application provides a really good and usable administration interface for managing structured content. django-content-editor extends Django's inlines mechanism with tools and an interface for managing heterogenous collections of content as are often necessary for content management systems. For example, articles may be composed of text blocks with images and videos interspersed throughout. Those content elements are called plugins.

django-tree-queries provides a smart way to efficiently fetch tree-shaped data in a relational database supporting Common Table Expressions.

Historical note

What we are calling plugins is called a content type in FeinCMS. This can be easily confused with Django's own contenttypes, therefore the name was changed for feincms3.

Using django-mptt or other tree libraries is possible with feincms3 as well if you don't want to use CTEs. Reimplementing the abstract page class with a different library should be straightforward.

1.1.3 The parts of feincms3

feincms3 has the following main parts:

- A base class for your own **pages** model if you want to use django-tree-queries to build a hierarchical page tree.
- Model **mixins** for common tasks such as building several navigation menus from a page tree, multilingual sites and selectable templates.
- A few ready-made **plugins** for rich text, images, oEmbed and template snippets.
- A HTML sanitization and **cleansing** function and a rich text widget building on html-sanitizer and django-ckeditor.
- Facilities for embedding **apps** through the admin interface, such as any interactive content (forms) or apps with subpages (e.g. an articles app).
- A **renderer** and associated helpers and template tags.
- **admin** classes for visualizing and modifying the tree hierarchy.
- Various utilities (**shortcuts** and **template tags**).

1.2 Prerequisites and installation

feincms3 runs on Python 3.8 or better. The minimum required Django version is 3.2. Database engine support is constrained by django-tree-queries use of recursive common table expressions. At the time of writing, this means that PostgreSQL, sqlite3 (>3.8.3) and MariaDB (>10.2.2) are supported. MySQL 8.0 should work as well, but is not being tested.

The best way to install feincms3 is:

```
pip install 'feincms3[all]'  # Quote the requirement to escape globbing
```

This installs all optional dependencies as required by the bundled rich text, image and oEmbed plugins. A more minimal installation can be selected by only running `pip install feincms3`.

1.3 Build your CMS

This guide shows step by step how to use the tools provided by feincms3 to build your own CMS.

Note: If you just want to quickly check out what feincms3 is capable of, have a look at the feincms3-example project. It shows how everything works together, but also uses advanced functionality which might be confusing to newcomers and is not necessary for smaller CMS projects.

1.3.1 Getting started

Install feincms3 and all recommended dependencies:

```
pip install feincms3[all]
```

Add the following settings:

```
INSTALLED_APPS = [
    ...
    "feincms3",
    "content_editor",
     # Optional, but not for this guide:
    "ckeditor",
    "imagefield",
]
```

1.3.2 Models

The page model and a few plugins could be defined as follows:

```
from django.db import models
from django.utils.translation import gettext_lazy as _

from content_editor.models import Region, create_plugin_base

from feincms3 import plugins
from feincms3.pages import AbstractPage


class Page(AbstractPage):
    regions = [
        Region(key="main", title=_("Main")),
    ]


PagePlugin = create_plugin_base(Page)


class RichText(plugins.richtext.RichText, PagePlugin):
    pass


class Image(plugins.image.Image, PagePlugin):
    pass
```

Note: The bundled rich text plugin (which we're going to integrate) uses *feincms3.cleanse.CleansedRichTextField* which always sends HTML through html-sanitizer. The default configuration of HTML sanitizer is really restrictive and removes images (besides other things such as normalizing the HTML and removing script tags etc.)

HTML copy-pasted from other sources (e.g. Word) is often messy. It is generally a good idea to sanitize HTML on the server side to prevent XSS attacks or even just the general uglyness that results from giving website editors too much

freedom.

We almost never allow embedding images, tables etc. into rich text elements on our sites. It is just too easy to add a 10MB JPEG or even a BMP file and scale it down to 50x50. Adding images as a separate plugin has other benefits too: No parsing of rich texts to replace images, it's much easier to e.g. create a lightbox, use the first image on the site as teaser image or whatever comes to your mind.

That being said, adding your own rich text plugin which allows whatever you want is quite straightforward and completely supported.

1.3.3 Rendering and templates

Here's an example how plugins could be rendered, `app.pages.renderer`:

```
from django.utils.html import format_html, mark_safe

from feincms3.renderer import RegionRenderer

from .models import Page, RichText, Image


renderer = RegionRenderer()
renderer.register(
    RichText,
    lambda plugin, context: mark_safe(plugin.text),
)
renderer.register(
    Image,
    lambda plugin, context: format_html(
        '<figure><img src="{}" alt=""/><figcaption>{}</figcaption></figure>',
        plugin.image.url,
        plugin.caption,
    ),
)
```

Of course if you'd rather let plugins use templates, do this:

```
from feincms3.renderer import template_renderer

renderer.register(
    Image,
    template_renderer("plugins/image.html"),
)
```

And the associated template:

```
<figure>
  <img src="{{ plugin.image.url }}" alt="{{ plugin.caption }}"/>
  {% if plugin.caption %}<figcaption>{{ plugin.caption }}</figcaption>{% endif %}
</figure>
```

The default image field also offers built-in support for thumbnailing and cropping with a PPOI (primary point of interest); have a look at the django-imagefield docs to find out how.

And a `pages/standard.html` template:

```
{% extends "base.html" %}

{% load feincms3 %}

{% block title %}{{ page.title }} - {{ block.super }}{% endblock %}

{% block content %}
  <main>
    <h1>{{ page.title }}</h1>
    {% render_region page_regions "main" %}
  </main>
{% endblock %}
```

It is recommended to add a utility as follows to the `app.pages.renderer` module:

```
def page_context(request, *, page):
    # page = page or page_for_app_request(request)
    page.activate_language(request)
    ancestors = list(page.ancestors().reverse())
    return {
        "page": page,
        "page_regions": renderer.regions_from_item(
            page,
            inherit_from=ancestors,
            timeout=30,
        ),
    }
```

1.3.4 Middleware

Note: The guide previously recommended to use a standard view for rendering pages. The problem is that you have to add a catch-all pattern to your URLconf which has some unwanted interactions e.g. with `i18n_patterns`. (All paths are resolvable but visiting them might still obviously generate 404 errors.) Because of this the guide now recommends using a middleware. Feel free to upgrade your code whenever you feel like it. Using *URLs and views* is documented and there's no reason to think it won't work in the future.

It is recommended to use a middleware to render pages. You're completely free to define your own middleware or even your own views and URLs. That being said, the `AbstractPage` class already has a `get_absolute_url` implementation which returns the page's `path`. If the Django app isn't mounted at `/` (this is possible but improbable) `get_absolute_url` automatically prepends the script prefix.

Note: `AbstractPage.get_absolute_url` still tries reversing `pages:page` and `pages:root` before falling back to the behavior described above.

A basic middleware module `app.pages.middleware` would look as follows:

```
from django.shortcuts import render
```

(continues on next page)

(continued from previous page)

```
from app.pages.models import Page
from app.pages.renderer import page_context


def page_middleware(get_response):
    def middleware(request):
        response = get_response(request)
        if response.status_code != 404:
            # Someone else already handled this request
            return response

        # path is the full path, path_info excludes the script prefix.
        if page := Page.objects.active().filter(path=request.path_info).first():
            return render(
                request,
                "pages/standard.html",
                page_context(request, page=page),
            )

        # No page found, fall back to the original 404 response
        return response

    return middleware
```

The `app.pages.middleware.page_middleware` middleware should be added at the end of `MIDDLEWARE`:

```
MIDDLEWARE = [
    ...
    "app.pages.middleware.page_middleware",
]
```

Note: Check *Root middleware for pages (feincms3.root)* later for more advanced middleware utilities.

Historical note

FeinCMS provided request and response processors and several ways how plugins (in FeinCMS: content types) could hook into the request-response processing. This isn't necessary with feincms3 – simply put the functionality into your own code.

1.3.5 Admin classes

Here's an example how the `app.pages.admin` module might look like:

```
from django.contrib import admin

from content_editor.admin import ContentEditor
from feincms3 import plugins
from feincms3.admin import TreeAdmin
```

(continues on next page)

(continued from previous page)

```
from app.pages import models


class PageAdmin(ContentEditor, TreeAdmin):
    list_display = ["indented_title", "move_column", "is_active"]
    prepopulated_fields = {"slug": ("title",)}
    raw_id_fields = ["parent"]

    inlines = [
        plugins.richtext.RichTextInline.create(models.RichText),
        plugins.image.ImageInline.create(models.Image),
    ]

    # fieldsets = ... (Recommended! No example here though. Note
    # that the content editor not only allows collapsed, but also
    # tabbed fieldsets -- simply add 'tabbed' to the 'classes' key
    # the same way you'd add 'collapse'.

    # class Media: ... (Add font-awesome from a CDN and nicely
    # looking buttons for plugins as is described in
    # django-content-editor's documentation -- search for
    # "plugin_buttons.js")


admin.site.register(models.Page, PageAdmin)
```

CHAPTER

TWO

GUIDES

These guides discuss key concepts and show how to approach common tasks. They do not have to be read in order and in general only build on the knowledge imparted in *Build your CMS*.

2.1 Writing plugins

Plugins for a given model extend an abstract base class created by django-content-editor's `create_plugin_base` function. A minimal example follows here:

```
from content_editor import Region, create_plugin_base

class Article(models.Model):
    regions = [Region(...)]
    title = models.CharField(...)

ArticlePlugin = create_plugin_base(Article)

class Text(ArticlePlugin):
    text = models.TextField()

class Download(ArticlePlugin):
    download = models.FileField(upload_to="downloads/")
    caption = models.CharField(blank=True, max_length=200)
```

The `create_plugin_base` creates an abstract model with the following fields and methods in the example above:

- `parent`: A foreign key to `Article`.
- `region`: A char field ready for holding the region's key it belongs to.
- `ordering`: An integer field which orders the list of content elements. The value of the ordering field should be treated as opaque in that you should not depend on exact values and gaps in the ordering field values.
- `get_queryset`: A classmethod without arguments which is used to fetch a queryset of plugin instances. If you have a plugin with a foreign key (not to `parent` but to other instances) it would probably be a really good idea to override this classmethod with one that adds a `select_related()` call.

Historical note

FeinCMS 1's `create_content_type` method could not be avoided because it added the dynamically created (concrete!) model to a few lists for bookkeeping.

By contrast using `create_plugin_base` is not strictly necessary. However, django-content-editor and by extension feincms3 assume a few properties which you'd have to replicate by hand such as the model fields, the `related_name` pattern etc.

2.2 Templates and regions

The build-your-CMS guide only used one region and one template. However, this isn't sufficient for many sites. Some sites have a moodboard region and maybe a sidebar region; some sites at least have a different layout on the home page and so on.

2.2.1 More regions

django-content-editor requires a `regions` attribute or property on the model containing a list of `Region` instances. The *Build your CMS* guide presented a page model with only a single region, `"main"`. It is of course possible to specify more regions:

```
class Page(AbstractPage):
    regions = [
        Region(key="main", title=_("Main")),
        Region(key="sidebar", title=_("Sidebar"), inherited=True),
    ]
```

Regions may also be marked as `inherited`. This means that pages deeper down in the tree may inherit content from some other page (normally the page's ancestors) in case the page region itself does not define any content.

The `page_detail` view presented in the guide also works with more than one region. However, for region inheritance to work you have to provide the pages whose content should be inherited yourself. There isn't much to do though, just add the `inherit_from` keyword argument:

```
# Inside a view or middleware:
page = ...
return render(
    request,
    "pages/standard.html",
    {
        "page": page,
        "page_regions": renderer.regions_from_item(
            page,
            inherit_from=page.ancestors().reverse(),
        ),
    },
)
```

`page.ancestors().reverse()` returns ancestors ordered from the page's parent to the root of the tree. We want pages to inherit content from their closest possible ancestors.

2.2.2 Making templates selectable

As written in the introduction above, sometimes a single template or layout isn't enough. Enter the *PageTypeMixin*:

```
from django.utils.translation import gettext_lazy as _
from content_editor.models import Region
from feincms3.applications import PageTypeMixin, TemplateType
from feincms3.pages import AbstractPage

class Page(AbstractPage, PageTypeMixin):
    TYPES = [
        TemplateType(
            key="standard",
            title=_("standard"),
            template_name="pages/standard.html",
            regions=[
                Region(key="main", title=_("Main")),
            ],
        ),
        TemplateType(
            key="with-sidebar",
            title=_("with sidebar"),
            template_name="pages/with-sidebar.html",
            regions=[
                Region(key="main", title=_("Main")),
                Region(key="sidebar", title=_("Sidebar"), inherited=True),
            ],
        ),
    ]
```

The `regions` attribute is provided by the `PageTypeMixin` and must be removed from the `Page` definition. Additionally, the `PageTypeMixin` provides a `type` property returning the currently selected page type. Instead of hard-coding the template we should now change the `page_detail` view to render the selected template, `page.type.template_name`:

```
# Inside a view or middleware:
page = ...
return render(
    request,
    page.type.template_name,
    {
        "page": page,
        "page_regions": renderer.regions_from_item(
            page,
            inherit_from=page.ancestors().reverse(),
        ),
    },
)
```

2.3 Redirects

The *feincms3.mixins.RedirectMixin* allows redirecting pages to other pages or arbitrary URLs. Inheriting the mixin adds two fields, the char field `redirect_to_url` and the self-referencing foreign key `redirect_to_page`:

```
from feincms3.mixins import RedirectMixin
from feincms3.pages import AbstractPage

class Page(AbstractPage, RedirectMixin):
    pass
```

At most one of `redirect_to_url` or `redirect_to_page` may be set, never both at the same time. The actual redirecting is not provided. This has to be implemented in the page view:

```
from django.http import HttpResponseRedirect

# Inside a view or middleware:
page = ...
if url := page.get_redirect_url():
    return HttpResponseRedirect(url)
# Default rendering continues here.
```

If you're using *Root middleware for pages (feincms3.root)* you can decorate your handler with `add_redirect_handler`.

2.4 Navigation generation recipes

This guide provides a few examples and snippets for generating the navigation for a site dynamically.

feincms3's *AbstractPage* model inherits the methods of `tree_queries.models.TreeNode`, notably `page.ancestors()` and `page.descendants()`. These methods and the attributes added by django-tree-queries, `tree_path` and `tree_depth` will prove useful for generating navigations.

feincms3 does not have a concrete page model and does not provide any tags to fetch page instances from the template. You'll have to provide the context variables yourself, preferrably by writing your own template tags.

This guide assumes that the concrete page model is available at `app.pages.models.Page`, and that you'll add a `menus` template tag library somewhere where it may be used by Django.

2.4.1 A simple main menu

Add the following template tag to the `menus.py` file:

```
from django import template
from app.pages.models import Page

register = template.Library()

@register.simple_tag
def main_menu():
    return Page.objects.with_tree_fields().active().filter(parent=None)
```

The template (where the `page` variable holds the current page):

```
{% load menus %}
{% main_menu as menu %}
<nav class="main-menu">
{% for p in menu %}
  <a
    {% if p.id in page.tree_path %}class="active"{% endif %}
    href="{{ p.get_absolute_url }}">{{ p.title }}</a>
{% endfor %}
</nav>
```

The `tree_path` attribute contains the list of all ancestors' primary keys including the key of the node itself. The `active` class is added to all menu entries that are either an ancestor of the current page or are the current page itself.

2.4.2 Breadcrumbs

Breadcrumbs for the current page are generated easily, without the help of a template tag. Ancestors are returned starting from the root node (again, the `page` variable holds the current page):

```
<nav class="breadcrumbs">
{% for ancestor in page.ancestors %}
  {% if forloop.last %}
    {{ ancestor.title }}
  {% else %}
    <a href="{{ ancestor.get_absolute_url }}">{{ ancestor.title }}</a>
    &gt;
  {% endif %}
{% endfor %}
</nav>
```

2.4.3 Main menu with two levels and meta navigation

For this example, the page class should also inherit the *feincms3.mixins.MenuMixin* with a `MENUS` value as follows:

```
from django.utils.translation import gettext_lazy as _
from feincms3.mixins import MenuMixin
from feincms3.pages import AbstractPage

class Page(AbstractPage, MenuMixin, ...):
    MENUS = (
        ("", "-"),  # Looks better than "None"
        ("main", _("main navigation")),
        ("meta", _("meta navigation")),
    )
```

Note: The menu choices should only use valid Python identifiers not starting with an underscore as values.

Let's write a template tag which returns all required nodes at once:

```
from collections import defaultdict
from django import template
from app.pages.models import Page

register = template.Library()

@register.simple_tag
def all_menus():
    menus = defaultdict(list)
    pages = Page.objects.with_tree_fields().active().exclude(
        menu=""
    ).extra(
        where=["tree_depth<=1"]
    )
    for page in pages:
        menus[page.menu].append(page)
    return menus
```

The template tag removes all pages that aren't added to a menu and filters for the first two levels in the tree. `tree_depth` is only available as an `.extra()` field, so you cannot use `.filter()` to do this.

Next, let's add a template filter which returns parents bundled together with their children:

```
@register.filter
def group_by_parent(iterable):
    parent = None
    children = []

    for element in iterable:
        if parent is None or element.tree_depth == parent.tree_depth:
            if parent:
                yield parent, children
                parent = None
                children = []

            parent = element
        else:
            children.append(element)

    if parent:
        yield parent, children
```

Now, a possible use of those two tags in the template looks as follows:

```
{% load menus %}
{% all_menus as menus %}

<nav class="nav-main">
{% for main, children in menus.main|group_by_parent %}
  <a
    {% if page and main.id in page.tree_path %}class="active"{% endif %}
    href="{{ main.get_absolute_url }}">{{ main.title }}</a>
    {% if children %}
    <nav>
```

(continues on next page)

(continued from previous page)

```
      {% for child in children %}
        <a
          {% if page and child.id in page.tree_path %}class="active"{% endif %}
          href="{{ child.get_absolute_url }}">{{ child.title }}</a>
      {% endfor %}
    </nav>
  {% endif %}
{% endfor %}
</nav>

{# ... and an analogous block for the meta menu, maybe without the children loop #}
```

2.5 Rendering

The default behavior of feincms3's renderer is to concatenate the rendered result of individual plugins into one long HTML string.

That may not always be what you want. This guide also describes a few alternative methods of rendering plugins that may or may not be useful.

2.5.1 Rendering plugins

The *feincms3.renderer.RegionRenderer* offers two fundamental ways of rendering content, string renderers and template renderers. The former simply return a string, the latter work similar to `{% include %}`.

String renderers

You may register a rendering function which returns a HTML string:

```
from django.utils.html import mark_safe
from feincms3.renderer import RegionRenderer
from app.pages.models import RichText

renderer = RegionRenderer()
renderer.register(
    RichText,
    lambda plugin, context: mark_safe(plugin.text)
)
```

Template renderers

Or you may choose to render plugins using a template:

```
from feincms3.renderer import template_renderer

renderer.register(
    Image,
    template_renderer("plugins/image.html"),
)
```

The configured template receives the plugin instance as `"plugin"`.

If you need more flexibility you may define your own plugin renderer:

```
from feincms3.renderer import render_in_context

def external_using_template(plugin, context):
    if "youtube" in plugin.url:
        template_name = "plugin/youtube.html"
    elif "vimeo" in plugin.url
        template_name = "plugin/vimeo.html"
    else:
        template_name = "plugin/external.html"
    return render_in_context(context, template_name, {"plugin": plugin})

renderer.register(
    External,
    external_using_template,
)
```

Note: You could also render the plugin using `render_to_string` but `render_in_context` has the advantage that it makes the surrounding context (using all context processors etc.) available, at least when using the `{% render_region %}` template tag.

Often, having the surrounding template context and the plugin instance available inside the template is enough. However, you might want to provide additional context variables. This can be achieved by specifying the `context` function. The function receives the plugin instance and the surrounding template context:

```
from feincms3.renderer import template_renderer

def plugin_context(plugin, context):
    return {
        "plugin": plugin,  # Recommended, but not required.
        "additional": ....
    }

renderer.register(
    Plugin,
    template_renderer("plugin/plugin.html", plugin_context),
)
```

Rendering individual plugins

Rendering individual plugin instances is possible using the `render_plugin` method. Except if you're using a non-standard `RegionRenderer` class used to encapsulate the fetching of plugins and rendering of regions you won't have to know about this method, but see below under *Grouping plugins into subregions*.

Regions instances

Because fetching plugins may be expensive (at least one database query per plugin type) it makes sense to avoid fetching plugins if there is a valid cached version. The *feincms3.renderer.RegionRenderer* which handles the specifics of rendering plugins belonging to specific regions has a method, `RegionRenderer.regions_from_item`, which automatically creates a lazily evaluated `content_editor.contents.Contents` instance.

The region renderer knows which plugins to load. It also supports inherited regions introduced in the *More regions* section of the *Templates and regions* guide.

The object returned by `regions_from_item` (and `regions_from_contents`) is an opaque object with the following interface:

- `regions` is the list of `content_editor.models.Region` objects.
- `render(region_key, context)` is a method which returns a single rendered region.

If `RegionRenderer.regions_from_item` received a `timeout` argument accesses to the interface above are automatically cached.

Note: Caching either works for all regions or for none at all.

Rendering regions in the template

The template requires the regions instance mentioned above so that regions can be rendered:

```
from .renderer import renderer

# Inside a view or middleware:
page = ...
...
return render(
    request,
    ...,
    {
        "page": page,
        "page_regions": renderer.regions_from_item(page, timeout=60),
    },
)
```

In the template you can now use the template tag:

```
{% load feincms3 %}

{% render_region page_regions "main" %}
```

Using the template tag is advantageous because it automatically provides the surrounding template context to individual plugins' renderers, meaning that they could for example access the `request` instance in a plugin template if e.g. an API key would be different for different URLs.

2.5.2 Grouping plugins into subregions

The `RegionRenderer` class supports rendering subregions differently. Plugins may be grouped automatically by their type or by some attribute they share.

Let's make an example. Assume that we want to group adjacent teaser elements. We have several teaser plugins but they all share the same `subregion` attribute value:

```
class ArticleTeaser(PagePlugin):
    article = models.ForeignKey(...)

class ProjectTeaser(PagePlugin):
    project = models.ForeignKey(...)
```

Next, we have to define a regions class which knows how to handle those teasers. The name of the handler has to match the subregion attribute exactly:

```
from feincms3.renderer import RegionRenderer

class SmartRegionRenderer(RegionRenderer):
    def handle_teasers(self, plugins, context):
        # Start the teasers element:
        yield '<div class="teasers">'
        for plugin in self.takewhile_subregion(plugins, "teasers"):
            # items is a deque, render the leftmost item:
            yield self.render_plugin(plugin, context)
        yield "</div>"

renderer = SmartRegionRenderer()
renderer.register(ArticleTeaser, ..., subregion="teasers")
renderer.register(ProjectTeaser, ..., subregion="teasers")
```

2.5.3 Grouping plugins into containers

The previous example added an `<div class="teasers">...</div>` wrapper element to a group of teasers. However, sometimes you want to allow some plugins to escape the containing element. In this case it may be useful to override the default subregions handler instead:

```
from django.utils.html import mark_safe

from feincms3.renderer import RegionRenderer, render_in_context

class FullWidthPlugin(PagePlugin):
    pass

class ContainerAwareRegionRenderer(RegionRenderer):
    def handle_default(self, plugins, context):
        content = [
```

(continues on next page)

(continued from previous page)

```
            self.render_plugin(plugin, context)
            for plugin in self.takewhile_subregion(plugins, "default")
        ]
        yield render_in_context(
            context, "subregions/default.html", {"content": content}
        )

    def handle_fullwidth(self, plugins, context):
        content = [
            self.render_plugin(plugin, context)
            for plugin in self.takewhile_subregion(plugins, "fullwidth")
        ]
        yield render_in_context(
            context, "subregions/fullwidth.html", {"content": content}
        )

# Instantiate renderer and register plugins
renderer = ContainerAwareRegionRenderer()
renderer.register(FullWidthPlugin, ..., subregion="fullwidth")

regions = renderer.regions_from_item(page)
output = regions.render(...)
```

2.5.4 Using marks

Some plugins may be usable inside several subregions. In this case you cannot simply set the `subregion`; you have to find another way.

One example may be a plugin which starts a collapsible region and which only supports text inside. Adding an image will automatically close the collapsible subregion as will adding another `CollapsibleRegionPlugin`.

```
class CollapsibleRegionPlugin(PagePlugin):
    title = models.CharField(
        _("title"),
        max_length=200,
        blank=True,
        help_text=_("Leave empty to explicitly finish a previously opened region."),
    )

class CollapsibleRegionRenderer(RegionRenderer):
    def handle_collapsible(self, plugins, context):
        collapsible = self.render_plugin(plugins.popleft(), context)
        content = [
            self.render_plugin(plugin, context)
            for plugin in self.takewhile_mark(plugins, "collapsible-content")
        ]
        yield from ...


renderer = CollapsibleRegionRenderer()
renderer.register(
```

(continues on next page)

(continued from previous page)

```
    CollapsibleRegionPlugin,
    lambda plugin, context: {
        "title": plugin.title, "is_collapsible": bool(plugin.title)
    },
    subregion="collapsible",
)
renderer.register(
    RichText,
    lambda plugin, context: mark_safe(plugin.text),
    marks={"collapsible-content"},
)
renderer.register(
    Image,
    # ...
)
```

2.5.5 Generating JSON

A different real-world example is generating JSON instead of HTML. This is possible with a custom `RegionRenderer` class too:

```
from feincms3.renderer import RegionRenderer

class JSONRegionRenderer(RegionRenderer):
    def render_region(self, *, region, contents, context):
        return [
            dict(
                self.render_plugin(plugin, context),
                type=plugin.__class__.__name__,
            )
            for plugin in contents[region.key]
        ]

        # Alternatively (In this case the ``type`` key above would have to be
        # provided by the renderers themselves):
        # return list(self.generate(self.contents[region], context))

def page_content(request, pk):
    page = get_object_or_404(Page, pk=pk)

    renderer = JSONRegionRenderer()
    renderer.register(
        RichText,
        lambda plugin, context: {"text": plugin.text},
    )
    renderer.register(
        Image,
        lambda plugin, context: {"image": request.build_absolute_uri(plugin.image.url)},
    )

    regions = renderer.regions_from_item(page, timeout=60)
```

(continues on next page)

(continued from previous page)

```
    return JsonResponse({
        "title": page.title,
        "content": regions.render("main", None),
    })
```

Note: A different method would have been to use lower-level methods from django-content-editor. A short example follows, however there's more left to do to reach the state of the example above such as caching:

```
from content_editor.contents import contents_for_items

renderers = {
    RichText: lambda plugin: {
        "text": plugin.text
    },
    Image: lambda plugin: {
        "image": request.build_absolute_uri(plugin.image.url)
    },
}
contents = contents_for_item(page, [RichText, Image])
data = [
    dict(
        renderers[plugin.__class__](plugin),
        type=plugin.__class__.__name__
    )
    for plugin in contents.main
]
# etc...
```

2.6 Cookie control

Some jurisidictions require the the users' consent before adding analytics scripts and tracking cookies. While it may be best to not use any analytics and tracking at all this may not be possible or even desirable in all circumstances.

Many solutions exist for adding a consent banner to the website. Some of those banners require loading JavaScript and other assets from external servers. This raises some questions because loading those scripts may also be seen as tracking already. It is certainly safer to implement a cookie control panel locally. It would be boring to start from scratch on each site.

This guide explains how to use feincms3-cookiecontrol.

2.6.1 Installation

Install the package:

```
venv/bin/pip install feincms3-cookiecontrol
```

Add `feincms3_cookiecontrol` to `INSTALLED_APPS` and add a custom location for the migrations to `MIGRATION_MODULES`. (feincms3-cookiecontrol cannot bundle migrations because it builds on django-translated-fields which has to know the list of available languages to create migrations.)

```
INSTALLED_APPS = [
    # ...
    "feincms3_cookiecontrol",
]

MIGRATION_MODULES = {
    "feincms3_cookiecontrol": "app.migrate.feincms3_cookiecontrol",
}
```

Note: You are of course free to use other values for `MIGRATION_MODULES`. The example above just works well. You'll have to create the `migrate` folder yourself and place an empty `__init__.py` file in there. Read the Django documentation for additional guidance.

Create and apply the initial migration:

```
python manage.py makemigrations feincms3_cookiecontrol
python manage.py migrate
```

Optionally load the fixture; it contains two cookie categories (essential and analytics) with short descriptions and translations (english, german, french and italian) while ignoring translations which aren't in use on your site:

```
python manage.py loaddata --ignorenonexistent f3cc-categories
```

Add the panel to the template, e.g. in `base.html` at the end of the <body> element:

```
<!doctype html>
<html>
  ...
  <body>
    ...
    {% load feincms3_cookiecontrol %}{% feincms3_cookiecontrol %}
  </body>
</html>
```

You'll have to add all tracking scripts yourself now.

2.6.2 Customize the appearance

The default colors of the control panel may not fit into your site. The best way to customize the appearance is to set a few CSS variables, e.g.:

```
#f3cc {
  --f3cc-accent-color: #abc;
}
```

2.6.3 Hiding the modify button

The default presentation of the panel is a fixed banner at the bottom of the viewport. Once any cookies have been accepted (essential cookies have to be accepted, e.g. the CSRF cookie) the banner is replaced by a single button which allows showing the control panel again.

You may want to suppress the button on some pages, for example on all pages except for the privacy policy.

A good way to achieve this follows.

Let's assume you're using page types as described in *Templates and regions*. Let's also assume that your privacy policy page uses the standard page type described in the guide:

```
class Page(AbstractPage, PageTypeMixin):
    TYPES = [
        TemplateType(
            key="standard",
            title=_("standard"),
            template_name="pages/standard.html",
            regions=[
                Region(key="main", title=_("Main")),
            ],
        ),
    ]
```

We will add an additional page type which can be used as a marker. Since we're using feincms3 apps be sure to read the *Introduction to apps* if you haven't done this already. You may also want to take a look at the *feincms3.root.passthru* reference.

```
class Page(AbstractPage, PageTypeMixin):
    TYPES = [
        TemplateType(
            key="standard",
            title=_("standard"),
            template_name="pages/standard.html",
            regions=[
                Region(key="main", title=_("Main")),
            ],
        ),
        ApplicationType(
            key="privacy-policy",
            title=_("privacy policy"),
            urlconf="feincms3.root.passthru",
            template_name="pages/standard.html",
            regions=[
```

(continues on next page)

(continued from previous page)

```
                Region(key="main", title=_("Main")),
            ],
        ),
    ]
```

Note: We cannot just use a new `TemplateType` because we **only** want to hide the button on all other pages if a privacy policy page actually exists!

Now you can extend the `page_context` helper:

```
from feincms3.root.passthru import reverse_passthru

def hide_modify_button(page):
    return bool(
        # We got a page instance
        page
        # The current page is not the privacy-policy page
        and page.type.key != "privacy-policy"
        # An active privacy policy page exists
        and reverse_passthru("privacy-policy", fallback=None)
    )

def page_context(request, *, page=None):
    ...
    return {
        ...
        "hide_modify_button": hide_modify_button(page),
    }
```

It's a bit involved but it's good to write defensive code.

Now you can use this additional variable in the template:

```
<!doctype html>
<html>
  ...
  <body>
    ...
    {% load feincms3_cookiecontrol %}
    {% feincms3_cookiecontrol hide_modify_button=hide_modify_button %}
  </body>
</html>
```

2.7 Meta and OpenGraph tags

The recommended way to add meta and open graph tags information to pages and other CMS objects is using feincms3-meta.

2.7.1 Installation and configuration

Install the package:

```
pip install feincms3-meta
```

Make the page model inherit the mixin:

```
from feincms3.pages import AbstractPage
from feincms3_meta.models import MetaMixin

class Page(AbstractPage, MetaMixin, ...):
    pass
```

If you define `fieldsets` on a `ModelAdmin` subclass, you may want to use the helper `MetaMixin.admin_fieldset()`, or maybe not.

Add settings (optional, but recommended):

```
META_TAGS = {
    "site_name": "My site",
    "title": "Default title",
    "description": (
        "The default description,"
        " maybe long."
    ),
    "image": "/static/app/logo.png",
    # "author": "...",
    "robots": "index,follow,noodp",
}

# Only for translations
INSTALLED_APPS.append("feincms3_meta")
```

2.7.2 Rendering

The dictionary subclass returned by `feincms3_meta.utils.meta_tags` can either be used as a dictionary, or rendered directly (its `__str__` method returns a HTML fragment):

```
from feincms3_meta.utils import meta_tags

# ...
return render(
    request,
    ...,
    {
        "page": page,
```

(continues on next page)

(continued from previous page)

```
        "page_regions": ...,
        ...
        "meta_tags": meta_tags([page], request=request),
    },
)
```

`meta_tags` also supports overriding or removing individual tags using keyword arguments. Falsy values are discarded, `None` causes the complete removal of the tag from the dictionary.

If you want to inherit meta tags from ancestors (or from other objects) provide more than one object to the `meta_tags` function:

```
ancestors = list(page.ancestors())
tags = meta_tags(request=request).add(*ancestors).add(page)
```

Since you may also need the ancestors when using regions which inherit content from the page's ancestors when they are empty it is recommended to put the meta tags generation into the `page_context` function described in *Build your CMS*. Note that `inherit_from` wants a reversed list of ancestors (from bottom to top) but `meta_tags` wants ancestors from top to bottom so that more specific values from lower in the page tree override their ancestors values:

```
def page_context(request, *, page):
    # page = page or page_for_app_request(request)
    page.activate_language(request)
    ancestors = list(page.ancestors())
    return {
        "page": page,
        "page_regions": renderer.regions_from_item(
            page,
            inherit_from=reversed(ancestors),
            timeout=30,
        ),
        "meta_tags": meta_tags(request=request).add(*ancestors).add(page),
    }
```

Note: If you want to further override meta tags e.g. in an application (see *Introduction to apps*) you may want to run the above function and reach into the context:

```
page = page_for_app_request(request)
context = page_context(request, page=page)

# Example: Article detail page
article = get_object_or_404(Article, ...)
context["meta_tags"].add(article)

return render(...)
```

2.8 URLs and views

Warning: As mentioned in *Build your CMS* the recommended way to render pages is using a middleware. It may be easier to use URLconf entries and views if you're integrating the CMS functionality into an existing site or when you want to use it only for a subtree. (A middleware may still be the way to go in this case.)

The default implementation of `AbstractPage.get_absolute_url` still expects the following URLconf entries and falls back to returning the path prefixed by the script prefix:

```
from django.shortcuts import redirect
from django.urls import path

from app.pages import views

app_name = "pages"
urlpatterns = [
    path("<path:path>/", views.page_detail, name="page"),
    path("", views.page_detail, name="root"),
]
```

Just to be on the safe side you may want to override `get_absolute_url` on your own page class (the default implementation does the same but catches `NoReverseMatch` exceptions and falls back to return the path prefixed with the script prefix):

```
class Page(AbstractPage, ...):
    def get_absolute_url(self):
        if self.path == "/":
            return reverse("pages:root")
        return reverse("pages:page", kwargs={"path": self.path.strip("/")})
```

A simple view might look as follows (building on the functionality from the introduction):

```
from django.shortcuts import get_object_or_404, render

from app.pages.models import Page
from app.pages.utils import page_context

def page_detail(request, path=None):
    page = get_object_or_404(Page.objects.active(), path=f"/{path}/" if path else "/")
    return render(request, "pages/standard.html", page_context(request, page=page))
```

CHAPTER

THREE

LANGUAGES AND SITES

3.1 Multilingual sites

This guide shows you how to implement several languages in a single site.

Note: If you also want to serve multiple sites in a single Django installation you should start here and continue with *Multisite setup*.

3.1.1 Making the page language selectable

Pages may come in varying languages. `LanguageMixin` helps with that. It adds a `language_code` field to the model which allows selecting the language based on `settings.LANGUAGES`. The first language is set as default:

```
from django.utils.translation import gettext_lazy as _
from feincms3.mixins import LanguageMixin
from feincms3.pages import AbstractPage

class Page(AbstractPage, LanguageMixin):
    class Meta(AbstractPage.Meta):
        unique_together = [("language_code", "translation_of")]
```

3.1.2 Activating the language

The `activate_language` method is the preferred way to activate the page's language for the current request. It runs `django.utils.translation.activate` and sets `request.LANGUAGE_CODE` to the value of `django.utils.translation.get_language`, the same things Django's `LocaleMiddleware` does.

Note that `activate` may fail and `get_language` might return a different language, however that's not specific to feincms3.

```
page = ...  # MAGIC! (or maybe get_object_or_404...)
page.activate_language(request)
...
```

Note: `page.activate_language` does not persist the language across requests as Django's `django.views.i18n.set_language` does. (`set_language` modifies the session and sets cookies.) That is mostly what you want though

since the page's language is tied to its URL.

3.1.3 Page tree tips

I most often add a root page per language, which means that the main navigation's `tree_depth` would be `1`, not `0`. The menu template tags described in *Navigation generation recipes* would also require an additional `.filter(language_code=django.utils.translation.get_language())` statement to only return pages in the current language.

A page tree might look as follows then:

```
Home (EN)
- About us
- News

Startseite (DE)
- Über uns
- Neuigkeiten

Page d'acceuil (FR)
- A propos de nous
- Actualité
```

By manually setting the slug of all root pages to their respective language code (e.g. Home (EN) has a URL of `/en/`, Startseite (DE) a URL of `/de/`) you can generate a navigation pointing to all sites in their respective language (assuming that the built-in template context processor `django.template.context_processors.i18n` is active):

```
<nav class="languages">
{% for code, name in LANGUAGES %}
  <a href="/{{ code }}/">{{ name }}</a>
{% endfor %}
</nav>
```

3.1.4 Deep links to pages in other languages

The navigation snippet above does not link translations directly but instead always shows the home page to visitors. It may be preferrable to define pages (and other CMS objects) as translations of each other so that it is possible to generate a navigation menu directly linking the same content in different languages directly.

feincms3 offers a built-in way to achieve this. Instead of inheriting the default *feincms3.mixins.LanguageMixin* inherit the *feincms3.mixins.LanguageAndTranslationOfMixin*. The latter provides an additional `translation_of` foreign key which allows linking pages in other languages to the page in the first language in the `LANGUAGES` setting's list. In the example above, you could specify that "Über uns" is the german translation of "About us", and "A propos de nous" the french translation of "About us". The *feincms3.mixins.LanguageAndTranslationOfMixin.translations()* method returns a list of all known translations. Together with the *translations()* template filter you can generate a navigation menu as follows (assuming that `object` is the current page):

```
{% load feincms %}
<nav class="languages">
{% for lang in page.translations.active|translations %}
  <a href="{% if lang.object %}{{ lang.object.get_absolute_url }}{% else %}/{{ lang.
↪code }}/{% endif %}">
```

(continues on next page)

(continued from previous page)

```
    {{ lang.name }}
  </a>
{% endfor %}
</nav>
```

LanguageAndTranslationOfMixin within feincms3.applications

The same should work for any CMS object inheriting *feincms3.mixins.LanguageAndTranslationOfMixin*, and should also work when used within a feincms3 app. (*Apps will be introduced later*.)

In this case it may be extra-important to wrap the object's call to *reverse_app()* in a block which overrides the active language so that the article is preferrably shown in a website with the matching language:

```
from django.utils.translation import override
from feincms3.applications import reverse_app

class Article(LanguageAndTranslationOfMixin, ...):
    class Meta:
        unique_together = [("language_code", "translation_of")]

    def get_absolute_url(self):
        with override(self.language_code):
            return reverse_app("articles", "detail", ...)
```

Generating the navigation menu for changing the language should preferrably link to the translated article and only fall back to the translated page's URL if no such article exists:

```
def article_detail(request, ...):
    page = page_for_app_request(request)
    page.activate_language(request)
    article = get_object_or_404(Article, ...)

    translations = {obj.language_code: obj for obj in page.translations().active()}
    translations.update(
        {obj.language_code: obj for obj in article.translations().active()}
    )

    # Use {% for lang in available_translations|translations %} ... {% endfor %}
    context = {"available_translations": translations.values()}
```

3.2 Multisite setup

feincms3-sites allows running a feincms3 site on several domains, with separate page trees etc. on each (if so desired). The same Django installation can serve several domains with different content without having to start a Django server per site as you'd have to if you were using `django.contrib.sites`. This also means that feincms3-sites isn't compatible with `django.contrib.sites` – you have to use one or the other.

Note: The simpler case of having exactly one site per language and one language per site is better supported by feincms3-language-sites.

3.2.1 Installation and configuration

Install the package:

```
pip install feincms3-sites
```

Inherit feincms3-sites's page model instead of the default feincms3 abstract page. The only difference is that this `AbstractPage` model has an additional `site` foreign key, and path uniqueness is enforced per-site:

```
from feincms3_sites.models import AbstractPage

class Page(AbstractPage, ...):
    pass
```

Add `FEINCMS3_SITES_SITE_MODEL = "feincms3_sites.Site"` to your `settings`.

Add `feincms3_sites` to `INSTALLED_APPS` and run migrations afterwards:

```
INSTALLED_APPS.append("feincms3_sites")
FEINCMS3_SITES_SITE_MODEL = "feincms3_sites.Site"
```

```
./manage.py migrate
```

Add `feincms3_sites.middleware.site_middleware` near the top of your `MIDDLEWARE` setting, in any case before `feincms3.applications.apps_middleware` if you're using it. The middleware either sets `request.site` to the current `feincms3_sites.models.Site` instance or raises a `Http404` exception.

The default behavior allows matching a single host. The advanced options fieldset in the administration panel of feincms3-sites allows specifying your own regex, allowing matching several hostnames. In this case you may also want to add `feincms3_sites.middleware.redirect_to_site_middleware` after the middleware mentioned above. If you're also using the `SECURE_SSL_REDIRECT` of Django's own `SecurityMiddleware` you have to add the `redirect_to_site_middleware` *before* `SecurityMiddleware`.

It is also possible to specify a default site. In this case, when no site's regex matches, the default site is selected instead as a fallback. The code does not prevent you from setting more than one site as the default but sites are deterministically ordered so the same site will always be selected.

3.2.2 Multisite support throughout your code

Since feincms3-sites 0.6 a contextvar automatically provides the current site when inside `site_middleware`. The default implementation of `Page.objects.active()` filters by the current site. When you're running queries on pages outside of a middleware you'll have to use the contextvar facility yourself by running your code inside a `with feincms3_sites.middleware.set_current_site(site):` block.

3.2.3 Default languages for sites

In some configurations it may be useful to specify a default language per site. In this case you should replace `django.middleware.locale.LocaleMiddleware` with `feincms3_sites.middleware.default_language_middleware`. This middleware has to be placed after the `site_middleware`.

3.3 Multisite Custom Site Model

Since feincms3-sites 0.13.1 you can use a custom site model instead of the default provided `feincms3_sites.Site`.

3.3.1 Configuration

First, make sure that you adjust your Django settings accordingly:

```
# FEINCMS3_SITES_SITE_MODEL = "feincms3_sites.Site"
FEINCMS3_SITES_SITE_MODEL = "myapp.Site"
```

From now on, Django will use your custom site model.

3.3.2 Implementing a Custom Site Model

Next, you have to implement your custom site model, by adding the following to your models, e.g.:

```
from feincms3_sites.models import AbstractSite
from feincms3_meta.models import MetaMixin

class CustomSite(AbstractSite, MetaMixin):
    name = models.CharField(max_length=100, blank=False, null=False)

    def __str__(self):
        return "{} ({})".format(self.name, self.host)
```

The above will add the additional `name` field to the site that is also used for display in dropdowns and lists in the site admin, e.g. `MySite (localhost:8000)`.

It will also inherit meta fields declared by the `MetaMixin` that can be used for providing default meta information to the pages associated with that site. (you will have to install feincms3-meta and enable the app in your Django settings to use it.)

Please note that the metadata from the site will not be added automatically, you will have to do this in your view, e.g.

```
from django.shortcuts import get_object_or_404, render, redirect

from feincms3_meta.utils import meta_tags

from .models import Page
from .renderer import renderer


def page_detail(request, path=None):
    """
    The view expects uri paths to be formed as such

    [[/<language-code>][/<path>]]

    in case of an empty uri path the view will redirect the user to

    /<site.default_language/
```

(continues on next page)

(continued from previous page)

```
    expecting a root page to exist that has the appropriate path

    otherwise, the provided uri path will be used and the system will
    try to retrieve a probably existing page via the ORM.
    """
    actual_site = request.site
    if not path:
        # Make sure that the user is redirected to the correct url.
        # We expect a root page with custom path /<default_language>/ to
        # exist here.
        redirect_path = "/{}/".format(actual_site.default_language)
        return redirect(redirect_path)

    actual_path = "/{}/".format(path)
    page = get_object_or_404(
        Page.objects.active(),
        path=actual_path
    )
    page.activate_language(request)
    ancestors = page.ancestors().reverse()
    meta_data_providers = [page] + list(ancestors) + [page.site]
    return render(
        request,
        page.type.template_name,
        {
            "page": page,
            "page_regions": renderer.regions_from_item(
                page,
                inherit_from=ancestors,
                timeout=60,
            ),
            "meta_tags": meta_tags(
                meta_data_providers,
                request=request,
                # The default site model doesn't have a name attribute, see
                # the custom site model above.
                site_name=page.site.name,
            )
        },
    )
```

Make sure that your custom site model gets registered with the Django ORM before your page model gets registered, otherwise there will be an exception telling you that the site model configured in `FEINCMS3_SITES_SITE_MODEL` does not exist.

Remember to update your migrations as well:

```
./manage.py makemigrations
./manage.py migrate
```

3.3.3 Implementing a Custom Site Model Admin

```
from django.contrib import admin
from django.utils.translation import gettext_lazy as _

from feincms3_sites.admin import DefaultLanguageListFilter
from feincms3_sites.admin import SiteAdmin
from feincms3_meta.models import MetaMixin

from .models import Site


@admin.register(Site)
class CustomSiteAdmin(SiteAdmin):
    list_display = [
        "name", "host", "default_language", "is_active", "is_default"
    ]

    list_filter = [
        "is_active", "host", "name", DefaultLanguageListFilter
    ]

    fieldsets = [
        (None, {
            "fields": [
                "name",
                "host",
                "default_language",
                "is_active",
                "is_default",
            ],
        }),
        (_("Advanced Settings"), {
            "fields": [
                "is_managed_re",
                "host_re",
            ],
            "classes": [
                "tabbed"
            ],
        }),
        MetaMixin.admin_fieldset()
    ]
```

By default, the admin edit/create page will display itself as a flat admin page, i.e. there will not be any tabs.

In comes the `django-content-editor` and the scripts and other media it provides.

So to have tabs on your custom site model's admin page, add the following to the `SiteAdmin`:

```
...

class CustomSiteAdmin(SiteAdmin):
```

(continues on next page)

(continued from previous page)

```
    ...

    class Media:
        css = {
            "all": [
                "content_editor/material-icons.css",
                "content_editor/content_editor.css",
            ]
        }
        js = [
            "admin/js/jquery.init.js",
            "content_editor/tabbed_fieldsets.js",
        ]
```

CHAPTER

FOUR

EMBEDDING APPS

feincms3 allows content managers to freely place pre-defined applications in the page tree. Examples for apps include forms, or a news app with archives, detail pages etc.

The apps documentation is meant to be read in order.

4.1 Introduction to apps

CMS plugins consist of static content. Backend code around plugins is restricted to rendering (except if you add a thing or two in your own views, of course).

However, wouldn't it be awesome if it were possible to add contact forms and even more complicated apps to the page tree through the CMS?

That's exactly what feincms3.applications are for.

Apps are defined by a list of URL patterns specific to this app. A simple contact form would probably only have a single URLconf entry (`r'^$'`), a news app would at least have two entries (the archive and the detail URL).

The activation of apps happens through a dynamically created URLconf module (probably the trickiest code in all of feincms3, *apps_urlconf()*). The *apps_middleware* assigns the module to `request.urlconf` which ensures that apps are available for resolving and URL reversing. No page code runs at all, control is directly passed to the app views.

Please note that apps do not have to take over the page where the app itself is attached. If the app does not have a URLconf entry for `r'^$'` the standard page rendering still happens. because of the recommended catch-all URLconf entry for pages at the end.

4.2 Adding a form builder app

The following example app uses form_designer to provide a forms builder integrated with the pages app described above. Apart from installing form_designer itself the following steps are necessary.

4.2.1 Extending the page model

Make the page model inherit `PageTypeMixin` and `LanguageMixin` and add a `TYPES` attribute to the class:

```
from feincms3.applications import PageTypeMixin
from feincms3.applications import ApplicationType, TemplateType
from feincms3.mixins import LanguageMixin
from feincms3.pages import AbstractPage

class Page(AbstractPage, AppsMixin, LanguageMixin, ...):
    # ...
    TYPES = [
        TemplateType(
            key="standard",
            title="...",
            regions=[Region(key="main", title="...")],
            # Available as page.type.template_name
            template_name="pages/standard.html",
        ),
        ApplicationType(
            key="forms",
            title=_("forms"),
            # Required: A module containing urlpatterns
            urlconf="app.forms",

            # If you want to add content, not just the form. Not required.
            regions=[Region(key="main", title="...")],

            # The "form" field on the page is required when
            # selecting the forms app
            required_fields=("form",),

            # Only necessary if you want to add more than one form to the
            # page tree (per language). Also helpful for finding a form's u
            # URL using reverse_app(f"forms-{form.pk}", "form")
            app_namespace=lambda page: f"{page.page_type}-{page.form_id}",
        ),
        # ...
    ]

    form = models.ForeignKey(
        "form_designer.Form",
        on_delete=models.SET_NULL,
        blank=True, null=True,
        verbose_name=_("form"),
    )
```

Note: The `LanguageMixin` is required, but if you have a site where there's only one language, you don't even have to show the `language_code` field in your administration panel. Simply make sure that the `LANGUAGES` setting contains only the one language and nothing else.

4.2.2 The application

Add the `app/forms.py` module itself. Note that since control is directly handed to the application view and no page view code runs you'll have to load the page instance yourself and do the necessary language setup and provide the page etc. to the rendering context. The best way to load the page instance responsible for the current app is by calling *feincms3.applications.page_for_app_request()*:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from django.urls import path

from feincms3.applications import page_for_app_request

from app.pages.renderer import renderer


def form(request):
    page = page_for_app_request(request)
    page.activate_language(request)

    context = {}

    if "ok" not in request.GET:
        form_class = page.form.form()

        if request.method == "POST":
            form = form_class(request.POST)

            if form.is_valid():
                # Discard return values from form processing.
                page.form.process(form, request)
                return HttpResponseRedirect("?ok")

        else:
            form = form_class()

        context["form"] = form

    context.update({
        "page": page,
        "page_regions": renderer.regions_from_item(
            page,
            inherit_from=page.ancestors().reverse(),
            timeout=60,
        )
    })

    return render(request, "form.html", context)


app_name = "forms"
urlpatterns = [
    path("", form, name="form"),
]
```

Add the required template:

```
{% extends "base.html" %}

{% load feincms3 %}

{% block content %}
  {% render_region page_regions 'main' %}

  {% if form %}
    <form method="post" action=".#form" id="form">
      {% csrf_token %}
      {{ form.as_p }}
      <button type="submit">Submit</button>
    </form>
  {% else %}
    <h1>Thank you!</h1>
  {% endif %}
{% endblock %}
```

Of course if you'd rather add another URL for the "thank you" page you're free to add a second entry to the `urlpatterns` list and redirect to this URL instead.

4.2.3 Outlook

The example above shows how to add a contact form at the end of the rest of the content. However, it would be quite easy to e.g. add a placeholder plugin which content managers can use to place the form somewhere in-between. An outline how this might be done follows:

The plugin model definition:

```
class Placeholder(PagePlugin):
    identifier = models.CharField(choices=[("form", "form")], ...)
```

The app:

```
def form(request):
    page = ...

    context = {}

    if "ok" not in request.GET:
        context.setdefault("placeholders", {})["form"] = form

    context.update({
        "page": page,
        "page_regions": renderer.regions(
            page, inherit_from=page.ancestors().reverse()),
    })

    return render(request, "form.html", context)
```

The rendering of the placeholder:

```
renderer.register_template_renderer(
    models.Placeholder,
    lambda plugin: "placeholder/{}.html".format(plugin.identifier),
    lambda plugin, context: {
        "plugin": plugin,
        "placeholder": context["placeholders"].get(plugin.identifier),
    },
)
```

The `placeholder/form.html` template:

```
<form method="post" action=".#form" id="form">
  {% csrf_token %}
  {{ form.as_p }}
  <button type="submit">Submit</button>
</form>
```

The rest of the steps is left as an exercise to the reader. The success message is missing, and missing is also what happens if the placeholder plugin hasn't been added to the page.

4.3 Apps and instances

Applications can be added to the tree several times. Django itself supports this through the differentiation between application namespaces and instance namespaces. feincms3 builds on this functionality.

The *feincms3.applications.apps_urlconf()* function generates a dynamic URLconf Python module including all applications in their assigned place and adding the `urlpatterns` from `ROOT_URLCONF` at the end (or returns the value of `ROOT_URLCONF` directly if there are no active applications).

4.3.1 Application namespaces and instance namespace

The application URLconfs are included using nested namespaces:

- The outer application namespace is `"apps"` by default.
- The outer instance namespace is `"apps-" + LANGUAGE_CODE`.
- The inner namespace is the app namespace, specified by the value of `app_name` in the apps' URLconf module. The string must correspond with the key of the *ApplicationType* instance in the `TYPES` list on the page model.
- The inner instance namespace is the same as the app namespace, except if you return a different value from the application type's `app_namespace` function.

Apps are contained in nested URLconf namespaces which allows for URL reversing using Django's `reverse()` mechanism. The inner namespace is the app itself, the outer namespace the language. (Currently the apps code depends on *LanguageMixin* and cannot be used without it.) *reverse_app()* hides a good part of the complexity of finding the best match for a given view name since apps will often be added several times in different parts of the tree, especially on sites with more than one language.

4.3.2 Reversing application URLs

The best way for reversing app URLs is by using *feincms3.applications.reverse_app()*. The method expects at least two arguments, a namespace and a viewname. The namespace argument also supports passing a list of namespaces which is useful in conjunction with the `app_namespace` option of applications.

reverse_app() first generates a list of viewnames and passes them on to *feincms3.applications.reverse_any()* (which returns the first viewname that can be reversed to a URL).

For the sake of an example let's assume that our site is configured with english, german and french as available languages and that we're trying to reverse the article list page, and that the current language (as returned by `get_language`) is german:

```
from feincms3.applications import reverse_app

# Inside a view or middleware:
articles_list_url = reverse_app("articles", "article-list")
...
```

The list of viewnames reversed is in order:

- `apps-de.articles.article-list`
- `apps-en.articles.article-list`
- `apps-fr.articles.article-list`

The german apps namespace comes first in the list. If the german part of the site does not contain an articles app, the reversing continues in all other languages.

If the namespace argument to *reverse_app()* was a list (or tuple), the list is even longer. Suppose that variants of the articles app may be added to the tree where only a single category is shown:

```
from feincms3.pages import AbstractPage
from feincms3.applications import PageTypeMixin
from feincms3.applications import ApplicationType, TemplateType

class Page(AbstractPage, PageTypeMixin, LanguageMixin, ...):
    TYPES = [
        TemplateType(
            key="standard",
            title="...",
            regions=[Region(key="main", title="...")],
            # Available as page.type.template_name
            template_name="pages/standard.html",
        ),
        ApplicationType(
            key="articles",
            title=_("Articles"),
            urlconf="app.articles.urls",
            app_namespace=lambda page: f"{page.page_type}-{page.category_id or 'all'}",
        ),
        ...
    ]

    category = models.ForeignKey(
        "articles.Category",
```

(continues on next page)

(continued from previous page)

```
        blank=True,
        null=True,
        ...
    )
```

In this case we might prefer the URL of a specific categories' articles app, but also be content with an articles app without a specific category:

```
reverse_app(
    [f"articles-{category.pk}", "articles"],
    "article-list",
)
```

The list of viewnames in this case is (assuming that the category has a `pk` value of 42):

- `apps-de.articles-42.article-list`
- `apps-de.articles.article-list`
- `apps-en.articles-42.article-list`
- `apps-en.articles.article-list`
- `apps-fr.articles-42.article-list`
- `apps-fr.articles.article-list`

As you can see `reverse_app` prefers apps in the current language to apps with the closer matching instance namespace.

Note: Some of the time Django's stock `reverse()` function works as well for reversing app URLs, e.g:

```
from django.urls import reverse

reverse("apps:articles:article-list")
```

However, it's still recommended to use `reverse_app`. `reverse` may not find apps because Django is content with the first match when searching for matching namespaces. Also, `reverse` may not find the best match in the presence of several app instances, be it because of several languages on the site or because of other factors.

4.3.3 Reversing URLs while preferring a specific language

Suppose that articles are written in a language, and `get_absolute_url` should prefer an app in this language to the same app in other languages. Since `reverse_app` automatically prefers the currently active language we use `override` to activate this language for the duration of the `reverse_app` call:

```
from django.utils.translation import override
from feincms3.applications import reverse_app
from feincms3.mixins import LanguageMixin

class Article(LanguageMixin):
    # ...

    def get_absolute_url(self):
        with override(self.language_code):
```

(continues on next page)

(continued from previous page)

```
            return reverse_app(
                "articles",
                "article-detail",
                kwargs={"slug": self.slug},

                # Pass a fallback value if you do not want to crash with a
                # NoReverseMatch exception if reversing fails. Maybe use a
                # better fallback value.though.
                fallback="/",
            )
```

Note: Previous versions recommended the use of *reverse_fallback()*. It's still available but not recommended anymore.

4.3.4 Reversing URLs outside the request-response cycle

Outside the request-response cycle, respectively outside *feincms3.applications.apps_middleware()*'s `request.urlconf` assignment, the reversing functions only use the URLconf module configured using the `ROOT_URLCONF` setting. In this case applications are impossible to find. However, all reversing functions support specifying the root URLconf module used for reversing:

```
from feincms3.applications import apps_urlconf, reverse_app

reverse_app("articles", "article-list", urlconf=apps_urlconf())
```

CHAPTER

FIVE

REFERENCE

5.1 Admin classes (feincms3.admin)

class feincms3.admin.**AncestorFilter**(*request*, *params*, *model*, *model_admin*)

Only show the subtree of an ancestor

By default, the first two levels are shown in the `list_filter` sidebar. This can be changed by setting the `max_depth` class attribute to a different value.

Usage:

```
class NodeAdmin(TreeAdmin):
    list_display = ("indented_title", "move_column", ...)
    list_filter = ("is_active", AncestorFilter, ...)

admin.site.register(Node, NodeAdmin)
```

lookups(*request*, *model_admin*)

Must be overridden to return a list of tuples (value, verbose value)

queryset(*request*, *queryset*)

Return the filtered queryset.

class feincms3.admin.**CloneForm**(**args*, ***kwargs*)

clean()

Hook for doing any extra form-wide cleaning after Field.clean() has been called on every field. Any ValidationError raised by this method will not be associated with a particular field; it will have a special-case association with the field named '__all__'.

property media

Return all media required to render the widgets on this form.

class feincms3.admin.**MoveForm**(**args*, ***kwargs*)

Allows making the node the left or right sibling or the first or last child of another node.

Requires the node to be moved as `obj` keyword argument.

clean()

Hook for doing any extra form-wide cleaning after Field.clean() has been called on every field. Any ValidationError raised by this method will not be associated with a particular field; it will have a special-case association with the field named '__all__'.

property media

Return all media required to render the widgets on this form.

class feincms3.admin.**TreeAdmin**(*model*, *admin_site*)

`ModelAdmin` subclass for managing models using django-tree-queries trees.

Shows the tree's hierarchy and adds a view to move nodes around. To use this class the two columns `indented_title` and `move_column` should be added to subclasses `list_display`:

```
class NodeAdmin(TreeAdmin):
    list_display = ("indented_title", "move_column", ...)

admin.site.register(Node, NodeAdmin)
```

get_queryset(*request*)

Return a QuerySet of all model instances that can be edited by the admin site. This is used by changelist_view.

get_urls()

Add our own `move` view.

indented_title(*instance*, *, *ellipsize=True*)

Use Unicode box-drawing characters to visualize the tree hierarchy.

move_column(*instance*)

Show a `move` link which leads to a separate page where the move destination may be selected.

5.2 Applications (feincms3.applications)

class feincms3.applications.**ApplicationType**(***kwargs*)

Application page type

Example usage:

```
from feincms3.applications import PageTypeMixin, TemplateType
from feincms3.pages import AbstractPage
from content_editor.models import Region

class Page(AbstractPage, PageTypeMixin)
    TYPES = [
        TemplateType(
            # Required arguments
            key="standard",
            title="Standard page",
            urlconf="path.to.urlconf.module",

            # Optional arguments
            template_name="pages/standard.html",
            regions=[
                Region(key="main", title="Main"),
            ],
            app_namespace=lambda page: ...,
```

(continues on next page)

(continued from previous page)

```
            # You may pass other arguments here, they will be available
            # on ``page.type`` as-is.
        ),
    ]
```

class feincms3.applications.**PageTypeMixin**(**args*, ***kwargs*)

The page class should inherit this mixin. It adds a `page_type` field containing the selected page type, and an `app_namespace` field which contains the instance namespace of the application, if the type of the page is an application type. The field is empty e.g. for template page types. Note that currently the *LanguageMixin* is a required dependency of *feincms3.applications*.

`TYPES` contains a list of page type instances, either *TemplateType* or *ApplicationType* and maybe others in the future. The configuration values are specific to each type, common to all of them are a key (stored in the `page_type` field) and a user-visible title.

Template types additionally require a `template_name` and a `regions` value.

Application types require a `urlconf` value and support the following options:

- `urlconf`: The path to the URLconf module for the application. Besides the `urlpatterns` list the module should probably also specify a `app_name`.
- `required_fields`: A list of page class fields which must be non-empty for the application to work. The values are checked in `PageTypeMixin.clean_fields`.
- `app_namespace`: A callable which receives the page instance as its only argument and returns a string suitable for use as an instance namespace.

Usage:

```
from content_editor.models import Region
from django.utils.translation import gettext_lazy as _
from feincms3.applications import PageTypeMixin
from feincms3.mixins import LanguageMixin
from feincms3.pages import AbstractPage

class Page(AbstractPage, PageTypeMixin, LanguageMixin):
    TYPES = [
        # It is recommended to always put a TemplateType type first
        # because it will be the default type:
        TemplateType(
            key="standard",
            title=_("Standard"),
            template_name="pages/standard.html",
            regions=[Region(key="main", title=_("Main"))],
        ),
        ApplicationType(
            key="publications",
            title=_("publications"),
            urlconf="app.articles.urls",
        ),
        ApplicationType(
            key="blog",
            title=_("blog"),
            urlconf="app.articles.urls",
```

(continues on next page)

(continued from previous page)

```
        ),
        ApplicationType(
            key="contact",
            title=_("contact form"),
            urlconf="app.forms.contact_urls",
        ),
        ApplicationType(
            key="teams",
            title=_("teams"),
            urlconf="app.teams.urls",
            app_namespace=lambda page: f"{page.page_type}-{page.team_id}",
            required_fields=["team"],
        ),
    ]
```

LANGUAGE_CODES_NAMESPACE = 'apps'

Override this to set a different name for the outer namespace.

clean_fields(*exclude=None*)

Checks that required fields are given and that an app namespace only exists once per site and language.

static fill_page_type_choices(*sender*, ***kwargs*)

Fills in the choices for `page_type` from the `TYPES` class variable. This method is a receiver of Django's `class_prepared` signal.

save(**args*, ***kwargs*)

Updates `app_namespace`.

property type

Returns the appropriate page type instance, either the selected type or the first type in the list of `TYPES` if no type is selected or if the type does not exist anymore.

class feincms3.applications.**TemplateType**(***kwargs*)

Template page type

Example usage:

```
from feincms3.applications import PageTypeMixin, TemplateType
from feincms3.pages import AbstractPage
from content_editor.models import Region

class Page(AbstractPage, PageTypeMixin)
    TYPES = [
        TemplateType(
            # Required arguments
            key="standard",
            title="Standard page",
            template_name="pages/standard.html",
            regions=[
                Region(key="main", title="Main"),
            ],

            # You may pass other arguments here, they will be available
            # on ``page.type`` as-is.
```

(continues on next page)

(continued from previous page)

```
        ),
    ]
```

feincms3.applications.**apps_middleware**(*get_response*)

This middleware must be put in `MIDDLEWARE`; it simply assigns the return value of *apps_urlconf()* to `request.urlconf`. This middleware should probably be one of the first since it has to run before any resolving happens.

feincms3.applications.**apps_urlconf**(*, *apps=None*)

Generates a dynamic URLconf Python module including all application page types in their assigned place and adding the `urlpatterns` from `ROOT_URLCONF` at the end. Returns the value of `ROOT_URLCONF` directly if there are no active application page types.

Since Django uses an LRU cache for URL resolvers, we try hard to only generate a changed URLconf when application URLs actually change.

The application URLconfs are put in nested namespaces:

- The outer application namespace is `apps` by default. This value can be overridden by setting the `LANGUAGE_CODES_NAMESPACE` class attribute of the page class to a different value. The instance namespaces consist of the `LANGUAGE_CODES_NAMESPACE` value with a language added at the end. As long as you're always using `reverse_app` you do not have to know the specifics.
- The inner namespace is the app namespace, where the application namespace is defined by the app itself (assign `app_name` in the same module as `urlpatterns`) and the instance namespace is defined by the application name (from `TYPES`).

Modules stay around as long as the Python (most of the time WSGI) process lives. Unloading modules is tricky and probably not worth it since the URLconf modules shouldn't gobble up much memory.

The set of applications can be overridden by passing a list of `(path, page_type, app_namespace, language_code)` tuples.

feincms3.applications.**page_for_app_request**(*request*, *, *queryset=None*)

Returns the current page if we're inside an app. Should only be called while processing app views. Will pass along exceptions caused by non-existing or duplicated apps (this should never happen inside an app because *apps_urlconf()* wouldn't have added the app in the first place if a matching page wouldn't exist, but still.)

Example:

```
def article_detail(request, slug):
    page = page_for_app_request(request)
    page.activate_language(request)
    instance = get_object_or_404(Article, slug=slug)
    return render(
        request,
        "articles/article_detail.html",
        {"article": article, "page": page},
    )
```

It is possible to override the queryset used to fetch a page instance. The default implementation simply uses the first concrete subclass of *PageTypeMixin*.

feincms3.applications.**reverse_any**(*viewnames*, *urlconf=None*, *args=None*, *kwargs=None*, *fallback=<object object>*, **fargs*, ***fkwargs*)

Try reversing a list of viewnames with the same arguments, and returns the first result where no `NoReverseMatch` exception is raised. Return `fallback` if it is provided and all viewnames fail to be reversed.

Usage:

```
url = reverse_any(
    ("blog:article-detail", "articles:article-detail"),
    kwargs={"slug": "article-slug"},
)
```

feincms3.applications.**reverse_app**(*namespaces*, *viewname*, **args*, *languages=None*, ***kwargs*)

Reverse app URLs, preferring the active language.

`reverse_app` first generates a list of viewnames and passes them on to `reverse_any`.

Assuming that we're trying to reverse the URL of an article detail view, that the project is configured with german, english and french as available languages, french as active language and that the current article is a publication, the viewnames are:

- `apps-fr.publications.article-detail`
- `apps-fr.articles.article-detail`
- `apps-de.publications.article-detail`
- `apps-de.articles.article-detail`
- `apps-en.publications.article-detail`
- `apps-en.articles.article-detail`

reverse_app tries harder returning an URL in the correct language than returning an URL for the correct instance namespace. The `fallback` keyword argument is supported too.

Example:

```
url = reverse_app(
    ("category-1", "blog"),
    "post-detail",
    kwargs={"year": 2016, "slug": "my-cat"},
)
```

feincms3.applications.**reverse_fallback**(*fallback*, *fn*, **args*, ***kwargs*)

Returns the result of `fn(*args, **kwargs)`, or `fallback` if the former raises a `NoReverseMatch` exception. This is especially useful for reversing app URLs from outside the app and you do not want crashes if the app isn't available anywhere.

The following two examples are equivalent, choose whichever you like best:

```
reverse_fallback(
    "/",
    lambda: reverse_app(
        ("articles",),
        "article-detail",
        kwargs={"slug": self.slug},
    ),
)

reverse_fallback(
    "/",
    reverse_app
    ("articles",),
```

(continues on next page)

(continued from previous page)

```
    "article-detail",
    kwargs={"slug": self.slug},
)
```

Note though that `reverse_app` supports directly specifying the fallback since 3.1.1:

```
reverse_app(
    ("articles",),
    "article-detail",
    kwargs={"slug": self.slug},
    fallback="/",
)
```

feincms3.templatetags.feincms3.**reverse_app**(*parser*, *token*)

Reverse app URLs, preferring the active language.

Usage:

```
{% load feincms3 %}
{% reverse_app 'blog' 'detail' [args] [kw=args] [fallback='/'] %}
```

`namespaces` can either be a list or a comma-separated list of namespaces. `NoReverseMatch` exceptions can be avoided by providing a `fallback` as a keyword argument or by saving the result in a variable, similar to `{% url 'view' as url %}` does:

```
{% reverse_app 'newsletter' 'subscribe-form' fallback='/newsletter/' %}
```

Or:

```
{% reverse_app 'extranet' 'login' as login_url %}
```

5.3 HTML cleansing (`feincms3.cleanse`)

HTML cleansing is by no means only useful for user generated content. Managers also copy-paste content from word processing programs, the rich text editor's output isn't always (almost never) in the shape we want it to be, and a strict allowlist based HTML sanitizer is the best answer I have.

class feincms3.cleanse.**CleansedRichTextField**(**args*, ***kwargs*)

This is a subclass of django-ckeditor's `RichTextField`. The recommended configuration is as follows:

```
CKEDITOR_CONFIGS = {
    "default": {
        "toolbar": "Custom",
        "format_tags": "h1;h2;h3;p;pre",
        "toolbar_Custom": [[
            "Format", "RemoveFormat", "-",
            "Bold", "Italic", "Subscript", "Superscript", "-",
            "NumberedList", "BulletedList", "-",
            "Anchor", "Link", "Unlink", "-",
            "HorizontalRule", "SpecialChar", "-",
            "Source",
```

(continues on next page)

(continued from previous page)

```
        ]],
    },
}

# Settings for feincms3.plugins.richtext.RichText
CKEDITOR_CONFIGS["richtext-plugin"] = CKEDITOR_CONFIGS["default"]
```

The corresponding `HTML_SANITIZERS` configuration for html-sanitizer would look as follows:

```
HTML_SANITIZERS = {
    "default": {
        "tags": {
            "a", "h1", "h2", "h3", "strong", "em", "p",
            "ul", "ol", "li", "br", "sub", "sup", "hr",
        },
        "attributes": {
            "a": ("href", "name", "target", "title", "id", "rel"),
        },
        "empty": {"hr", "a", "br"},
        "separate": {"a", "p", "li"},

        # Additional default settings not listed here.
    },
}
```

At the time of writing those are the defaults of html-sanitizer, so you don't have to do anything.

If you want or require a different cleansing function, simply override the default with `CleansedRichTextField(cleanse=your_function)`. The cleansing function receives the HTML as its first and only argument and returns the cleansed HTML.

clean(*value*, *instance*)

Convert the value's type and run validation. Validation errors from to_python() and validate() are propagated. Return the correct value if no error is raised.

`feincms3.cleanse.`**cleanse_html**(*html*)

Pass ugly HTML, get nice HTML back.

5.4 Embedding (`feincms3.embedding`)

Embedding videos and other 3rd party content without oEmbed.

`feincms3.embedding.`**embed**(*url*, *, *handlers=[embed_youtube, embed_vimeo]*)

Run a selection of embedding handlers and return the first value, or `None` if URL couldn't be processed by any handler.

You could write your own handler converting the URL argument into a plain old anchor element or maybe even *feincms3.plugins.external.oembed_html()* if you wanted to fall back to oEmbed.

`feincms3.embedding.`**embed_vimeo**(*url*)

Return HTML for embedding Vimeo videos or `None`, if argument isn't a Vimeo link.

The Vimeo <iframe> is wrapped in a element.

feincms3.embedding.**embed_youtube**(*url*)

Return HTML for embedding YouTube videos or `None`, if argument isn't a YouTube link.

The YouTube <iframe> is wrapped in a element.

5.5 Inline CKEditor (feincms3.inline_ckeditor)

class feincms3.inline_ckeditor.**InlineCKEditorField**(**args*, ***kwargs*)

This field uses an inline CKEditor 4 instance to edit HTML. All HTML is cleansed using html-sanitizer.

The default configuration of both `InlineCKEditorField` and HTML sanitizer only allows a heavily restricted subset of HTML. This should make it easier to write CSS which works for all possible combinations of content which can be added through Django's administration interface.

The field supports the following keyword arguments to alter its configuration and behavior:

- `cleanse`: A callable which gets messy HTML and returns cleansed HTML.
- `ckeditor`: A CDN URL for CKEditor 4.
- `config`: Change the CKEditor 4 configuration. See the source for the current default.
- `config_name`: Alternative way of configuring the CKEditor. Uses the `FEINCMS3_CKEDITOR_CONFIG` setting.

clean(*value*, *instance*)

Run the cleaned form value through the `cleanse` callable

contribute_to_class(*cls*, *name*, ***kwargs*)

Add a `get_*_excerpt` method to models which returns a de-HTML-ified excerpt of the contents of this field

deconstruct()

Act as if we were a `models.TextField`. Migrations do not have to know that's not 100% true.

formfield(***kwargs*)

Ensure that forms use the `InlineCKEditorWidget`

5.6 Mixins (feincms3.mixins)

class feincms3.mixins.**LanguageAndTranslationOfMixin**(**args*, ***kwargs*)

This object not only has a language, it may also be a translation of another object.

clean_fields(*exclude=None*)

Implement the following validation rules:

- Objects in the primary language cannot be the translation of another object
- Objects in other languages can only reference objects in the primary language (this is automatically verified by Django because we're using `limit_choices_to`)

translations()

Return a queryset containing all translations of this object

This method can be called on any object if translations have been defined at all, you do not have to fetch the object in the primary language first.

class feincms3.mixins.**LanguageMixin**(**args*, ***kwargs*)

Pages may come in varying languages. `LanguageMixin` helps with that.

activate_language(*request*)

`activate()` the page's language and set `request.LANGUAGE_CODE`

class feincms3.mixins.**MenuMixin**(**args*, ***kwargs*)

The `MenuMixin` is most useful on pages where there are menus with differing content on a single page, for example the main navigation and a meta navigation (containing contact, imprint etc.)

static fill_menu_choices(*sender*, ***kwargs*)

Fills in the choices for `menu` from the `MENUS` class variable. This method is a receiver of Django's `class_prepared` signal.

class feincms3.mixins.**RedirectMixin**(**args*, ***kwargs*)

The `RedirectMixin` allows adding redirects in the page tree.

clean_fields(*exclude=None*)

Ensure that redirects are configured properly.

get_redirect_url()

Return the URL for the redirect, if a redirect is configured

class feincms3.mixins.**TemplateMixin**(**args*, ***kwargs*)

It is sometimes useful to have different templates for CMS models such as pages, articles or anything comparable. The `TemplateMixin` provides a ready-made solution for selecting django-content-editor `Template` instances through Django's administration interface.

Warning: You are encouraged to use the PageTypeMixin and TemplateType from *feincms3.applications* instead.

static fill_template_key_choices(*sender*, ***kwargs*)

Fills in the choices for `menu` from the `MENUS` class variable. This method is a receiver of Django's `class_prepared` signal.

property regions

Return the selected template instances' `regions` attribute, falling back to an empty list if no template instance could be found.

property template

Return the selected template instance if the `template_key` field matches, or falls back to the first template in `TEMPLATES`.

5.7 Pages (feincms3.pages)

class feincms3.pages.**AbstractPage**(**args*, ***kwargs*)

Short version: If you want to build a CMS with a hierarchical page structure, use this base class.

It comes with the following fields:

- `parent`: (a nullable tree foreign key) and a `position` field for relatively ordering pages. While it is technically possible for `position` to be 0, e.g., data bulk imported from another CMS, it is not recommended, as the `save()` method will override values of 0 if you manipulate pages using the ORM.
- `is_active`: Boolean field. The `save()` method ensures that inactive pages never have any active descendants.
- `title` and `slug`
- `path`: The complete path to the page, starting and ending with a slash. The maximum length of path (1000) should be enough for everyone (tm, famous last words). This field also has a unique index, which means that MySQL with its low limit on unique indexes will not work with this base class. Sorry.
- `static_path`: A boolean which, when `True`, allows you to fill in the `path` field all by yourself. By default, `save()` ensures that the `path` fields are always composed of a concatenation of the parent's `path` with the page's own `slug` (with slashes of course). This is especially useful for root pages (set `path` to `/`) or, when building a multilingual site, for language root pages (i.e. `/en/`, `/de/`, `/pt-br/` etc.)

clean_fields(*exclude=None*)

Check for path uniqueness problems.

get_absolute_url()

Return the page's absolute URL

If path is `/`, reverses `pages:root` without any arguments, alternatively reverses `pages:page` with an argument of `path`. Note that this `path` is not the same as `self.path` – slashes are stripped from the beginning and the end of the string to make building an URLconf more straightforward.

If any `reverse()` call fails it falls back to returning `self.path` prefixed with the script prefix which is `/` in the standard case.

save(*self*, *...*, *save_descendants=None*)

Saves the page instance, and traverses all descendants to update their `path` fields and ensure that inactive pages (`is_active=False`) never have any descendants with `is_active=True`.

By default, descendants are only updated when any of `is_active` and `path` change. This can be overridden by either forcing updates using `save_descendants=True` or skipping them using `save_descendants=False`.

class feincms3.pages.**AbstractPageQuerySet**(*model=None*, *query=None*, *using=None*, *hints=None*)

Defines a single method, `active`, which only returns pages with `is_active=True`.

active()

Return only active pages

This function is used in *apps_urlconf()* and is the recommended way to fetch active pages in your code as well.

feincms3.pages.**path_with_script_prefix**(*path*)

Return `path` prefixed with the current script prefix

5.8 Plugins (feincms3.plugins)

Historical note

The content types in FeinCMS had ways to process requests and responses themselves, the `.process()` and `.finalize()` methods. feincms3 plugins do not offer this. The feincms3 way to achieve the same thing is by using *apps* or by adding the functionality in your own views (which are much simpler than the view in FeinCMS was).

5.8.1 External

Uses the Noembed oEmbed service to embed (almost) arbitrary URLs. Depends on requests.

class feincms3.plugins.external.**External**(**args*, ***kwargs*)

External content plugin

class feincms3.plugins.external.**ExternalInline**(*parent_model*, *admin_site*)

class feincms3.plugins.external.**NoembedValidationForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *instance=None*, *use_required_attribute=None*, *renderer=None*)

Tries fetching the oEmbed code for the given URL when cleaning form data

This isn't active by default. If you want to validate URLs you should use the following snippet:

```
from app.pages import models
from feincms3 import plugins

class SomeAdmin(...):
    inlines = [
        ...
        plugins.external.ExternalInline.create(
            model=models.External,
            form=plugins.external.NoembedValidationForm,
        ),
        ...
    ]
```

clean()

Hook for doing any extra form-wide cleaning after Field.clean() has been called on every field. Any ValidationError raised by this method will not be associated with a particular field; it will have a special-case association with the field named '__all__'.

property media

Return all media required to render the widgets on this form.

feincms3.plugins.external.**oembed_html**(*url*, ***kwargs*)

Wraps *oembed_json()*, but only returns the HTML part of the OEmbed response.

The return value is always either a HTML fragment or an empty string.

feincms3.plugins.external.**oembed_json**(*url*, *, *cache_failures=True*, *force_refresh=False*, *params=None*)

Asks Noembed for the embedding HTML code for arbitrary URLs. Sites supported include YouTube, Vimeo, Twitter and many others.

Successful embeds are always cached for 30 days.

Failures are cached if `cache_failures` is `True` (the default). The durations are as follows:

- Connection errors are cached 60 seconds with the hope that the connection failure is only transient.
- HTTP errors codes and responses in an unexpected format (no JSON) are cached for 24 hours.

The return value is always a dictionary, but it may be empty.

feincms3.plugins.external.**render_external**(*plugin*, ***kwargs*)

Render the plugin, embedding it in the appropriate markup for Foundation's responsive-embed element

The HTML embed code is generated using *oembed_html()*. Maybe you want to take a look at *feincms3.embedding* for a less versatile but much faster alternative.

5.8.2 HTML

Plugin providing a simple textarea where raw HTML, CSS and JS code can be entered.

Most useful for people wanting to shoot themselves in the foot.

class feincms3.plugins.html.**HTML**(**args*, ***kwargs*)

Raw HTML plugin

class feincms3.plugins.html.**HTMLInline**(*parent_model*, *admin_site*)

Just available for consistency, absolutely no difference to a standard `ContentEditorInline`.

feincms3.plugins.html.**render_html**(*plugin*, ***kwargs*)

Return the HTML code as safe string so that it is not escaped. Of course the contents are not guaranteed to be safe at all

5.8.3 Images

Provides an image plugin with support for setting the primary point of interest. This is very useful especially when cropping images. Depends on django-imagefield.

class feincms3.plugins.image.**Image**(**args*, ***kwargs*)

Image plugin

class feincms3.plugins.image.**ImageInline**(*parent_model*, *admin_site*)

Image inline

feincms3.plugins.image.**render_image**(*plugin*, ***kwargs*)

Return a simple, unscaled version of the image

5.8.4 Rich text

class feincms3.plugins.richtext.**RichTextInline**(*parent_model*, *admin_site*)

feincms3.plugins.richtext.**render_richtext**(*plugin*, ***kwargs*)

Return the text of the rich text plugin as a safe string (`mark_safe`)

5.8.5 Snippets

Plugin for including template snippets through the CMS

class feincms3.plugins.snippet.**Snippet**(**args*, ***kwargs*)

Template snippet plugin

static fill_template_name_choices(*sender*, ***kwargs*)

Fills in the choices for `template_name` from the `TEMPLATES` class variable. This method is a receiver of Django's `class_prepared` signal.

classmethod register_with(*renderer*)

This helper registers the snippet plugin with a `TemplatePluginRenderer` while adding support for template-specific context functions. The templates specified using the `TEMPLATES` class variable may contain a callable which receives the plugin instance and the template context and returns a context dictionary.

class feincms3.plugins.snippet.**SnippetInline**(*parent_model*, *admin_site*)

Snippet inline does nothing special, it simply exists for consistency with the other feincms3 plugins

feincms3.plugins.snippet.**render_snippet**(*plugin*, ***kwargs*)

Renders the selected template using `render_to_string`

5.8.6 Deprecated plugins

Rich text support using django-ckeditor

Provides a rich text area whose content is automatically cleaned using a very restrictive allowlist of tags and attributes.

Depends on django-ckeditor and html-sanitizer.

class feincms3.plugins.old_richtext.**RichText**(**args*, ***kwargs*)

Rich text plugin

To use this, a django-ckeditor configuration named `richtext-plugin` is required. See the section *HTML cleansing* for the recommended configuration.

class feincms3.plugins.old_richtext.**RichTextInline**(*parent_model*, *admin_site*)

The only difference with the standard `ContentEditorInline` is that this inline adds the `feincms3/plugin-ckeditor.css` file which adjusts the width of the django-ckeditor widget inside the content editor.

feincms3.plugins.old_richtext.**render_richtext**(*plugin*, ***kwargs*)

Return the text of the rich text plugin as a safe string (`mark_safe`)

5.9 Regions (feincms3.regions)

Warning: This module is deprecated. Use *feincms3.renderer.RegionRenderer* instead.

class feincms3.regions.**Regions**(*, *contents*, *renderer*, *cache_key=None*, *timeout=None*)

Regions uses content_editor.contents.Contents and the feincms3.renderer.TemplatePluginRenderer to convert a list of plugins into a rendered representation, most often a HTML fragment.

This class may also be instantiated directly but using the factory methods (starting with from_) below is probably more comfortable.

classmethod from_contents(*contents*, *, *renderer*, ***kwargs*)

Create and return a regions instance using the bare minimum of a contents instance and a renderer. Additional keyword arguments are forwarded to the regions constructor.

classmethod from_item(*item*, *, *renderer*, *inherit_from=None*, *timeout=None*, ***kwargs*)

Create and return a regions instance for an item (for example a page, an article or anything else managed by django-content-editor).

The item's plugins are determined by what is registered with the renderer. The plugin instances themselves are loaded lazily, and loading every time can be avoided completely by specifying a timeout.

generate(*items*, *context*)

Inspects all items in the region for a subregion attribute and passes control to the subregions' respective rendering handler, named handle_<subregion>. If subregion is not set or is falsy handle_default is invoked instead. This method raises a KeyError exception if no matching handler exists.

You probably want to call this method when overriding render.

handle_default(*items*, *context*)

Renders items from the queue using the renderer instance as long as the items either have no subregion attribute or whose subregion attribute is an empty string.

render(*region*, *context=None*)

Main function for rendering.

Starts the generator and assembles all fragments into a safe HTML string.

feincms3.regions.**cached_render**(*fn*)

Decorator for Regions.render methods implementing caching behavior

feincms3.regions.**matches**(*item*, *, *plugins=None*, *subregions=None*)

Checks whether the item matches zero or more constraints.

plugins should be a tuple of plugin classes or None if the type shouldn't be checked.

subregions should be set of allowed subregion attribute values or None if the subregion attribute shouldn't be checked at all. Include None in the set if you want matches to succeed also when encountering an item without a subregion attribute.

feincms3.templatetags.feincms3.**render_region**(*context*, *regions*, *region*, ***kwargs*)

Render a single region. See *RegionRenderer* for additional details.

Usage:

```
{% render_region regions "main" %}
```

5.10 Renderer (feincms3.renderer)

exception feincms3.renderer.**PluginNotRegistered**

Exception raised when encountering a plugin which isn't known to the renderer.

class feincms3.renderer.**RegionRenderer**

The region renderer knows how to render single plugins and also complete regions.

The basic usage is to instantiate the region renderer, register plugins and render a full region with it.

handle(*plugins*, *context*)

Runs the `handle_<subregion>` handler for the head of the `plugins` deque.

This method requires that a matching handler for all values returned by `self.subregion()` exists.

You probably want to call this method when overriding the rendering of a complete region.

handle_default(*plugins*, *context*)

Renders plugins from the queue as long as there are plugins belonging to the `default` subregion.

marks(*plugin*)

Return the marks of a plugin instance

plugins()

Return a list of all registered plugin classes

regions_from_contents(*contents*, ***kwargs*)

Return an opaque object encapsulating `content_editor.contents.Contents` and the logic required to render them.

All you need to know is that the return value has a `regions` attribute containing a list of regions and a `render` method accepting a region key and a context instance.

regions_from_item(*item*, */*, ***, *inherit_from=None*, *timeout=None*, ***kwargs*)

Return an opaque object, see *regions_from_contents()*

Automatically caches the return value if `timeout` is truthy. The default cache key only takes the `item`'s class and primary key into account. You may have to override the cache key by passing `cache_key` if you're doing strange^Wadvanced things.

register(*plugin*, *renderer*, */*, *subregion='default'*, *marks={'default'}*)

Register a plugin class

The renderer is either a static value or a function which always receives two arguments: The plugin instance and the context. When using `{% render_region %}` and the Django template language the context will be a Django template `Context` (or even `RequestContext`) instance.

The two optional keyword arguments' are:

- `subregion: str = "default"`: The subregion for this plugin as a string or as a callable accepting a single plugin instance. A matching `handle_<subregion>` callable has to exist on the region renderer instance or rendering **will** crash loudly.
- `marks: Set[str] = {"default"}`: The marks of this plugin. Marks only have the meaning you assign to them. Marks are preferrable to running `isinstance` on plugin instances especially when using the same region renderer class for different content types (e.g. pages and blog articles).

register_string_renderer(*plugin*, *renderer*)

Backwards compatibility for `TemplatePluginRenderer`. It is deprecated, don't use in new code.

register_template_renderer(*plugin*, *template_name*, *context=<function default_context>*)

Backwards compatibility for `TemplatePluginRenderer`. It is deprecated, don't use in new code.

render_plugin(*plugin*, *context*)

Render a single plugin using the registered renderer

render_plugin_in_context(*plugin*, *context=None*)

Backwards compatibility for `TemplatePluginRenderer`. It is deprecated, don't use in new code.

render_region(***, *region*, *contents*, *context*)

Render one region.

subregion(*plugin*)

Return the subregion of a plugin instance

takewhile_mark(*plugins*, *mark*)

Yield all plugins from the head of the `plugins` deque as long as their marks include `mark`.

takewhile_subregion(*plugins*, *subregion*)

Yield all plugins from the head of the `plugins` deque as long as their subregion equals `subregion`.

class feincms3.renderer.**TemplatePluginRenderer**(**args*, ***kwargs*)

TemplatePluginRenderer is deprecated, use *RegionRenderer*.

feincms3.renderer.**default_context**(*plugin*, *context*)

Return the default context for plugins rendered with a template, which simply is a single variable named `plugin` containing the plugin instance.

feincms3.renderer.**render_in_context**(*context*, *template*, *local_context=None*)

Render using a template rendering context

This utility avoids the problem of `render_to_string` requiring a `dict` and not a full-blown `Context` instance which would needlessly burn CPU cycles.

feincms3.renderer.**template_renderer**(*template_name*, *local_context=<function default_context>*, */*)

Build a renderer for the region renderer which uses a template (or a list of templates) and optionally a local context function. The context contains the site-wide context variables too when invoked via `{% render_region %}`

5.11 Root middleware for pages (feincms3.root)

5.11.1 Page middleware (feincms3.root.middleware)

The guide recommends using a middleware for the feincms3 pages app. This module offers helpers and utilities to reduce the amount of code you have to write. The reason why this module is called `root` is that the page app's mountpoint has to be the Python app's mountpoint when using this. If that's not the case you may want to write your own *URLs and views*.

Example code for using this module (e.g. `app.pages.middleware`):

```
from django.shortcuts import render
from feincms3.root.middleware import add_redirect_handler, create_page_if_404_middleware
```

(continues on next page)

(continued from previous page)

```
from app.pages.models import Page
from app.pages.utils import page_context

# The page handler receives the request and the page.
# ``add_redirect_handler`` wraps the handler function with support for the
# RedirectMixin.
@add_redirect_handler
def handler(request, page):
    return render(request, page.type.template_name, page_context(request, page=page))

# This is the middleware which you want to add to ``MIDDLEWARE`` as
# ``app.pages.middleware.page_if_404_middleware``. The middleware should be
# added in the last position except if you have a very good reason not to
# do this.
page_if_404_middleware = create_page_if_404_middleware(
    # queryset=Page.objects.active() works too (if .active() doesn't use
    # get_language or anything similar)
    queryset=lambda request: Page.objects.active(),

    handler=handler,
)
```

feincms3.root.middleware.**add_redirect_handler**(*handler*)

Wrap the page handler in a redirect mixin handler

feincms3.root.middleware.**create_page_if_404_middleware**(*, *queryset*, *handler*, *language_code_redirect=False*)

Create a middleware for handling pages

This utility is there for your convenience, you do not have to use it. The returned middleware already handles returning non-404 responses as-is, fetching a page instance from the database and calling a user-defined handler on success. It optionally also supports redirecting requests to the root of the app to a language-specific landing page.

Required arguments:

- `queryset`: A page queryset or a callable accepting the request and returning a page queryset.
- `handler`: A callable accepting the request and a page and returning a response.

Optional arguments:

- `language_code_redirect` (`False`): Redirect visitor to the language code prefix (e.g. `/en/`, `/de-ch/`) if request path equals the script prefix (generally `/`) and no active page for `/` exists.

5.11.2 Passthru page apps (`feincms3.root.passthru`)

The idea of this module is to allow tagging pages to allow programmatically determing the URL of pages which are often linked to, e.g. privacy policy or imprint pages.

Create an application type:

```
TYPES = [
    ...
    ApplicationType(
```

(continues on next page)

(continued from previous page)

```
        key="imprint",
        title=_("imprint"),
        urlconf="feincms3.root.passthru",
        template_name="pages/standard.html",
        regions=[Region(key="main", title=_("Main"))],
    ),
]
```

Reverse the URL of the page (if it exists):

```
# Raise NoReverseMatch on failure
reverse_passthru("imprint")

# Fallback
reverse_passthru("imprint", fallback="/en/imprint/")

# Outside the request-response cycle
reverse_passthru("imprint", urlconf=apps_urlconf())
```

`feincms3.root.passthru.`**`reverse_passthru`**(*namespace*, ***kwargs*)

Reverse a passthru app URL

Raises `NoReverseMatch` if page could not be found.

5.12 Shortcuts (`feincms3.shortcuts`)

For me, the most useful part of Django's generic class based views is the template name generation and the context variable naming for list and detail views, and also the pagination.

The rest of the CBV is less flexible than I'd like them to be, i.e. integrating forms on detail pages can be a hassle.

Because of this, `render_list` and `render_detail`.

`feincms3.shortcuts.`**`render_detail`**(*request*, *object*, *context=None*, ***, *template_name_suffix='_detail'*)

Render a single item

Usage example:

```
def article_detail(request, slug):
    article = get_object_or_404(Article.objects.published(), slug=slug)
    return render_detail(
        request,
        article,
    )
```

An additional context dictionary is also supported, and specifying the template name suffix too.

The `Article` instance in the example above is passed as `object` AND `article` (lowercased model name) into the template.

`feincms3.shortcuts.`**`render_list`**(*request*, *queryset*, *context=None*, ***, *model=None*, *paginate_by=None*, *template_name_suffix='_list'*)

Render a list of items

Usage example:

```
def article_list(request, ...):
    queryset = Article.objects.published()
    return render_list(
        request,
        queryset,
        paginate_by=10,
    )
```

You can also pass an additional context dictionary and/or specify the template name suffix. The query parameter `page` is hardcoded for specifying the current page if using pagination.

The queryset (or the page if using pagination) are passed into the template as `object_list` AND `<model_name>_list`, i.e. `article_list` in the example above.

`feincms3.shortcuts.`**`template_name`**(*model*, *template_name_suffix*)

Given a model and a template name suffix, return the resulting template path:

```
>>> template_name(Article, "_detail")
"articles/article_detail.html"
>>> template_name(User, "_form")
"auth/user_form.html"
```

5.13 Template tags (`feincms3.templatetags.feincms3`)

`feincms3.templatetags.feincms3.`**`maybe_target_blank`**(*href*, ***, *attributes='target="_blank" rel="noopener"'*)

Return the value of `attributes` if the first argument isn't a first party link (as determined by *is_first_party_link()*)

Usage:

```
<a href="{{ url }}" {% maybe_target_blank url %}>...</a>
```

`feincms3.templatetags.feincms3.`**`render_region`**(*context*, *regions*, *region*, ***kwargs*)

Render a single region. See *RegionRenderer* for additional details.

Usage:

```
{% render_region regions "main" %}
```

`feincms3.templatetags.feincms3.`**`reverse_app`**(*parser*, *token*)

Reverse app URLs, preferring the active language.

Usage:

```
{% load feincms3 %}
{% reverse_app 'blog' 'detail' [args] [kw=args] [fallback='/'] %}
```

`namespaces` can either be a list or a comma-separated list of namespaces. `NoReverseMatch` exceptions can be avoided by providing a `fallback` as a keyword argument or by saving the result in a variable, similar to `{% url 'view' as url %}` does:

```
{% reverse_app 'newsletter' 'subscribe-form' fallback='/newsletter/' %}
```

Or:

```
{% reverse_app 'extranet' 'login' as login_url %}
```

feincms3.templatetags.feincms3.**translations**(*iterable*)

Return a list of dictionaries, one for each language in `settings.LANGUAGES`. An example follows:

```
[
    {"code": "en", "name": "English", "object": <instance>},
    {"code": "de", "name": "German", "object": None},
    # ...
]
```

The filter accepts anything you throw at it. "It" should be an iterable of objects having a `language_code` property however, or anything non-iterable (such as `None`). The filter always returns a list of all languages in `settings.LANGUAGES` but the `object` key's value will always be `None` if the data is unusable.

5.14 Utils (feincms3.utils)

Note: The utils module is meant purely for feincms3's internal use. Utilities may be added and removed without prior warning and without a deprecation period.

If you depend on some functionality from this module copy the code into your project (according to the very permissive license of course).

class feincms3.utils.**ChoicesCharField**(**args*, ***kwargs*)

`models.CharField` with choices, which makes the migration framework always ignore changes to `choices`, ever.

deconstruct()

Return enough information to recreate the field as a 4-tuple:

- The name of the field on the model, if contribute_to_class() has been run.
- The import path of the field, including the class, e.g. django.db.models.IntegerField. This should be the most portable version, so less specific may be better.
- A list of positional arguments.
- A dict of keyword arguments.

Note that the positional or keyword arguments must contain values of the following types (including inner values of collection types):

- None, bool, str, int, float, complex, set, frozenset, list, tuple, dict
- UUID
- datetime.datetime (naive), datetime.date
- top-level classes, top-level functions - will be referenced by their full import path
- Storage instances - these have their own deconstruct() method

This is because the values here must be serialized into a text format (possibly new Python code, possibly JSON) and these are the only types with encoding handlers defined.

There's no need to return the exact way the field was instantiated this time, just ensure that the resulting field is the same - prefer keyword arguments over positional ones, and omit parameters with their default values.

feincms3.utils.**is_first_party_link**(*url*, *, *first_party_hosts=None*)

Return whether an URL is a first-party link or not.

First parties are defined by `ALLOWED_HOSTS` and can be overridden by passing an alternative list of hosts. The wildcard `["*"]` isn't recognized.

NOTE! `first_party_hosts` should not contain port numbers even if using a non-standard port, the same is true for Django's `ALLOWED_HOSTS` setting.

One template tag is available to help with ensuring off-site links open in a new window (if you need this…). The template tag does not allow specifying the list of first party hosts (it always uses `ALLOWED_HOSTS`):

```
{% load feincms3 %}
<a href="{{ url }}" {% maybe_target_blank url %}>text</a>
```

feincms3.utils.**validation_error**(*error*, *, *field*, *exclude*, ***kwargs*)

Return a validation error that is associated with a particular field if it isn't excluded from validation.

See https://github.com/django/django/commit/e8c056c31 for some background.

CHAPTER

SIX

PROJECT LINKS

6.1 Change log

6.1.1 Next version

- Added Django 4.1b1 to the CI matrix.
- Specified a custom plugin button for the old richtext plugin.

6.1.2 3.6 (2022-05-12)

- Fixed the `APPEND_SLASH` handling to also use `request.path_info`, not `request.path`.
- Added support for embedding YouTube shorts when using *feincms3.embedding*.
- Added autogenerated API documentation for the template tags and the `old_richtext` plugin to the docs.

6.1.3 3.5 (2022-04-11)

- Changed the feincms3 code to not generate feincms3 deprecation warnings (only in the testsuite, for compatibility). Changed `Snippet.register_with` to use the new region renderer API. Changed `Regions` to use `render_plugin` instead of `render_plugin_in_context`.
- Changed `TreeAdmin.indented_title` to ellipsize super-long titles by default.
- Added a system check which verifies that page types have distinct keys.
- Imported the `old_richtext` module to `feincms3.plugins` as long as it is available.
- Changed the linked CKEditor version to 4.18.0.
- Added `APPEND_SLASH` handling to the middleware created by *feincms3.root.middleware.create_page_if_404_middleware()*. This has to be done explicitly because valid page paths aren't resolvable when using a middleware.

6.1.4 3.4 (2022-03-10)

- Added a system check verifying that the appropriate `unique_together` value is set when using the `LanguageAndTranslationOfMixin`.
- Added a system check for the `app_name` value of application URLconf modules.
- Added a system check for the values of `MenuMixin.MENUS`.
- Slowly start deprecating the *`TemplateMixin`*. (It probably won't go away for a long time.)

6.1.5 3.3 (2022-03-03)

- Changed the root middleware to not act on 404 responses generated by views, only on 404 responses generated by resolver failures.
- Removed two deprecated `PageTypeMixin` properties (`application` and `app_instance_namespace`).
- Added a `{% maybe_target_blank url %}` template tag which helps with adding `target="_blank" rel="noopener"` to the template when opening third party links, if you really need this.

6.1.6 3.2 (2022-03-01)

- Added a `fallback` keyword argument to *`reverse_app()`*, *`reverse_any()`* and `reverse_passthru()` which offers an easy way to avoid the `NoReverseMatch` exception when a fallback value is actually OK. This can be used to avoid the *`reverse_fallback()`* wrapper.
- Upgraded a few code patterns in the docs to use recommended functionality.
- Changed the guides to nudge people towards using middleware instead of catch-all URLconf patterns.
- Renamed `feincms3.incubator.root` to `feincms3.root.middleware` and renamed `feincms3.incubator.root_passthru` to `feincms3.root.passthru`, thereby making them officially supported.

6.1.7 3.1 (2022-03-01)

- Changed the link color in the inline CKEditor to be readable in dark mode.
- Added a direct dependency on django-js-asset (django-content-editor already depends on it so it's nothing new) and fixed a deprecation warning in our usage.
- Added a system check for `ApplicationType` instances which errors out if the referenced URLconf modules cannot be imported.
- Added a few style resets for CKEditor 4 popups so that it works better in the Django admin's dark mode.
- Added a fallback to *`feincms3.pages.AbstractPage.get_absolute_url()`* which returns the page's path prefixed with the script prefix if reversing the URL fails.
- Changed the "Build your CMS" guide to recommend a middleware instead of URLs and views.
- Added the `feincms3.incubator.root` and `feincms3.incubator.root_passthru` modules which support using middleware to render pages.
- Changed the linked CKEditor version to 4.17.2.

6.1.8 3.0 (2022-02-09)

- Introduced a new *feincms3.renderer.RegionRenderer* infrastructure which merges and replaces *feincms3.regions* and *feincms3.renderer.TemplatePluginRenderer*. The new module supports other template engines and handles subregions without polluting models with attributes, making it possible to use several renderers in the same project with differing subregion configurations.

6.1.9 2.1 (2022-01-13)

- Exposed the list of content editor regions on *Regions* as `regions`. Raised the minimum django-content-editor version to 6.0.

6.1.10 2.0 (2022-01-03)

- Added pre-commit.
- Dropped compatibility with Python < 3.8, Django < 3.2.
- Changed the linked CKEditor version to 4.17.1.
- Fixed the move form CSS when used with Django 4.0. It's not consistent yet but better.

6.1.11 1.0 (2021-12-03)

- Fixed a Python 3.8-ism.
- Added a `params` parameter to *feincms3.plugins.external.oembed_json()* which allows overriding values sent to the oEmbed provider.
- Added a `force_refresh` parameter to *feincms3.plugins.external.oembed_json()* which allows forcibly refreshing the cached oEmbed data.
- Added a threadlocal cache to `apps_urlconf` which allows calling `apps_urlconf` several times without producing database queries over and over.
- Added Python 3.10 to the CI.
- Changed `LanguageAndTranslationOfMixin.translation_of` to use a `TreeNodeForeignKey` so that the hierarchy is shown when using a dropdown.
- Raised the minimum version of django-content-editor to 5.0.

6.1.12 0.94 (2021-09-29)

- Inline CKEditor: Dropped the admin jQuery dependency for real.
- Started using pyupgrade for the Python code.
- Added Django 4.0a1 to the CI matrix.
- Added a way to configure the inline CKEditor through Django settings.

6.1.13 0.93 (2021-09-20)

- Changed *feincms3.embedding.embed_youtube()* to append `?rel=0` to the YouTube embed URL which should hopefully suppress recommendations when the embedded video ends.
- **Slightly backwards incompatible**: Dropped the Noembed validation from the default `feincms3.plugins.external` admin inline. Renamed the (undocumented!) `ExternalForm` to `NoembedValidationForm`.
- Raised the versions of required dependencies to recent versions, especially django-tree-queries to include a fix for the upcoming Django 4.0.
- Inline CKEditor: Changed the CDN URL to reference CKEditor 4.16.2.
- Inline CKEditor: Changed the JavaScript code to not hard-depend on jQuery.

6.1.14 0.92 (2021-06-09)

- Raised the minimum version of django-content-editor to 5.0a3 to take advantage of the bundled Material Icons library. Added default icon specifications to all plugins' inlines.
- Fixed a bug where `feincms3.plugins.richtext` wasn't available when `django-ckeditor` wasn't installed despite no longer depending on it anymore.

6.1.15 0.91 (2021-05-28)

Inline CKEditor widget

This release deprecates the django-ckeditor integration of feincms3 and officially introduces a new rich text widget which uses the inline mode of CKEditor 4. It looks better and avoids the scrollable text area inside the (scrollable!) content editor.

- Moved the inline CKEditor out of the incubator. It is a good idea and we should commit to supporting it.
- **BACKWARDS INCOMPATIBLE**: The *feincms3.plugins.richtext* plugin has been replaced by a widget using an inline CKEditor instance. The new field looks better and doesn't depend on django-ckeditor anymore. The `CKEDITOR_CONFIGS` setting from django-ckeditor isn't used anymore either, so if you reconfigured the rich text editor you'll have to update the configuration again. The old plugin is still available as *feincms3.plugins.old_richtext* for the time being.
- **BACKWARDS INCOMPATIBLE**: The *feincms3.cleanse* module has been deprecated. The inline CKEditor includes the cleansing functionality too.
- Inline CKEditor: Updated the CKEditor CDN URL to include the 4.16.1 patch release.
- Removed django-ckeditor from the `all` extra of feincms3. This means that installing `feincms3[all]` doesn't automatically install django-ckeditor anymore.

6.1.16 0.90 (2021-04-27)

This release contains a few backwards-incompatible changes which are the result of efforts to produce a better foundation and fix oversights towards a 1.0 release of feincms3.

Page types

Introduced the concept of page types. Merged the functionality of `TemplateMixin` and `AppsMixin` into a new `PageTypeMixin` and removed `AppsMixin`. Editors do not have to choose a template anymore when activating an app. The latter overrides the former selection anyway. Also, this allows using a custom selection of regions per application.

The following steps should be followed to upgrade existing sites:

- Create an automatic migration for the pages app.
- Edit the generated migration; create the `page_type` field first, and insert a `RunSQL` migration with the following SQL next: `UPDATE pages_page SET page_type=CASE WHEN application<>'' THEN application ELSE template_key END`.
- Ensure that the `app_instance_namespace` is renamed to `app_namespace` using a `RenameField` operation.
- Remove `template_key` from any code and replace `application` with `page_type` in the model admin configuration.
- Convert the entries in your `TEMPLATES` list to `TemplateType` instances, convert `APPLICATIONS` to `ApplicationType` instances and add both to a new `TYPES` class-level list. Note that those applications do not have *any* regions by default at all.
- The `.template` attribute of page classes does not exist any longer, to access e.g. the `template_name` replace `page.template.template_name` with `page.type.template_name`.
- Replace uses of `page.application` with `page.page_type`, `page.app_instance_namespace` with `page.app_namespace`. Properties mapping the former to the latter will stay in place for a release or two but they are already deprecated.

Other backwards-incompatible changes

- Added `alternative_text` and `caption` fields to the image and the external plugin. Made both plugins prefer the caption in `__str__`.
- Dropped the django-versatileimagefield-based image plugin.
- Removed the shims in `feincms3.apps`.
- Standardized `max_length` values of `CharField` instances.
- Changed the snippet plugin to no longer try to render templates not in the `TEMPLATES` list. This means that you can just remove templates from `TEMPLATES` and not worry about database contents referencing templates which could have been removed in the meantime in the base case.

Minor changes

- Tried out a web-based translation platform. It wasn't exactly a big success, but we gained a few translations. Thanks to all contributors!
- Added a system check for page subclasses without the appropriate ordering definition.
- Changed the docs so that `AbstractPage` always comes before mixins so that `AbstractPage`'s `Meta` properties are actually inherited by default.
- Changed the docs to recommend `HttpResponseRedirect` for the *feincms3.mixins.RedirectMixin* redirect, not the `redirect` shortcut. The latter may crash if the `redirect_to_url` doesn't look like a URL.
- Removed useless fallbacks.
- Fixed background colors in the move form to work with Django admin's dark mode.
- Added a `feincms3/static-path-style.js` script which automatically reduces the opacity of the path field unless the path is defined manually.
- Introduced an experimental inline CKEditor field.
- Raised the minimum django-content-editor version to 4.1 to take advantage of `content_editor.models.Type`.

6.1.17 0.41 (2020-11-28)

- Switched from Travis CI to GitHub Actions.
- Dropped the custom CKEditor activation JavaScript, django-ckeditor does all we need already.

6.1.18 0.40 (2020-09-30)

- Changed the move form styling (hide the radio inputs and use background colors, stripes to visualize the tree structure better.
- Added a warning when trying to move a node but there are no valid targets.
- Fixed the move form widget in the responsive layout.
- Avoided removing the parent node from the move form when moving the first child.
- Added a `get_redirect_url` to the *RedirectMixin* which returns the target URL or `None`.
- Added the *feincms3.utils.is_first_party_link()* utilty.

6.1.19 0.39 (2020-09-25)

- **BACKWARDS INCOMPATIBLE**: `AbstractPageManager` has been removed. You should subclass the *feincms3.pages.AbstractPageQuerySet* instead and use the queryset's `.as_manager(with_tree_fields=True)` classmethod to generate a manager which adds tree fields to select queries by default. If you didn't use the `AbstractPageManager` in your code directly you don't have to do anything.
- Started requiring `django-tree-queries>=0.4.1`.
- Completely reworked the page move form; allow directly specifying the new position.

6.1.20 0.38.1 (2020-09-23)

- The `AbstractPageManager.active()` method has been moved to a new *feincms3.pages.AbstractPageQuerySet*. If subclassing the queryset you should re-create the page manager using `pages.AbstractPageManager.from_queryset(<your new subclass>)`.
- Made *render_in_context()* create its own `Context` if the context passed is `None`.

6.1.21 0.37 (2020-09-10)

- Changed *feincms3.applications.page_for_app_request()* to only use active pages by default. This change should mostly not change anything since *apps_urlconf()* and therefore *apps_middleware()* only add active applications anyway.
- Upgraded prettier and ESLint to recent versions.
- Added some code to embed videos from YouTube and Vimeo without requiring oEmbed.
- Dropped compatibility with Python 3.5.

6.1.22 0.36 (2020-08-07)

- Switched from `url()` to `re_path()` in `apps_urlconf()` to avoid deprecation warnings.
- Removed the limitation that apps could not have descendants in a page tree. There may be valid use cases for this, especially if an apps' URLconf module does not handle *all* paths.

6.1.23 0.35 (2020-07-28)

- **(not yet) BACKWARDS INCOMPATIBLE** Moved the `feincms3.apps` module to *feincms3.applications*. The reason for this change is that Django 3.2 will start autodiscovering app configs and therefore automatically loads the `.apps` submodule of all entries in `INSTALLED_APPS`. This leads to a crash when the `.apps` module contains models (such as our `AppsMixin`). `feincms3.apps` isn't populated from Django 3.2 upwards because of this.
- Fixed an infinite recursion crash when referencing pages using `on_delete=SET_NULL`
- Added a `LanguageAndTranslationOfMixin` which not only allows defining the language of objects but also defining objects to be translations of other objects.
- Added a `|translations` filter to the template tag library. Added a section about generating a language selector containing deep links to the *multilingual sites guide* guide.
- Added Travis CI jobs for Django 3.1b1 and Python 3.8.
- Renamed the main branch to `main`.
- Removed all arguments to `super()` since we're Python 3-only.
- Dropped workarounds for the removal of `django.utils.six` and `python_2_unicode_compatible` from the testsuite. They were only required for our dependencies, not for feincms3 itself.

6.1.24 0.34 (2020-06-05)

- Removed mentions of Python 2 compatibility in the docs.
- Allowed using `render_list` with lists, not only querysets.
- Dropped compatibility with Django<2.2 in accordance with the official Django releases support policy.
- Replaced `url()` with `re_path()` which avoids a few deprecation warnings.

6.1.25 0.33 (2019-12-16)

- Changed `Regions`' `cache_key` argument handling to allow disabling caching by returning a falsy value.
- Added the `feincms3.renderer.render_in_context` utility.
- Verified compatibility with Django 3.0.
- Made the `TemplateMixin.template` property fall back to the first template in `TEMPLATES` if the specific template could not be found or does not exist.
- Fixed another path uniqueness validation problem where pages having descendants with static paths could not be saved.

6.1.26 0.32 (2019-09-20)

- Changed `app_instance_namespace` to `blank=True` to make it clear what the default value is.
- Fixed a possible path uniqueness problem with descendants with static paths.
- Dropped Python 3.4 compatibility.

6.1.27 0.31 (2019-05-14)

- Added copying of `handler400`, `handler403`, `handler404` and `handler500` from `ROOT_URLCONF` to the URLconf module created by `apps_urlconf`.

Removed all deprecated features

- The `AppsMiddleware` alias for `apps_middleware` has been removed.
- The `feincms3.incubator` module has has been removed including subrenderers.
- The `depth` and `cte_path` attributes of `AbstractPage` have been removed. Those helped with the transition from django-cte-forest to django-tree-queries almost one year ago.
- `TemplatePluginRenderer.regions()` and `feincms3.renderer.Regions` are replaced by `feincms3.regions.Regions`. Region timeouts must be specified when instantiating the `feincms3.regions.Regions` object and cannot be specified when rendering individual regions anymore.
- The `feincms3_apps` and `feincms3_renderer` template tag libraries have been replaced by a single `feincms3` tag library.

6.1.28 0.30 (2019-03-18)

- Fixed overflowing tree structure boxes in the `TreeAdmin`.
- Switched to emitting `DeprecationWarning` warnings not `Warning`, even though their visibility sucks.
- Added a `languages` argument to `reverse_app` which allows overriding languages and their order.
- Made `TreeAdmin` and `MoveForm` only require that the default manager is a `TreeQuerySet` and not that the model itself also extends `TreeNode`.
- Made `plugin_ckeditor.js`'s dependency on `django.jQuery` explicit. This is necessary for Django 2.2's new `Media.merge` algorithm.

6.1.29 0.29 (2019-02-07)

- Deprecated the `feincms3_apps` and `feincms3_renderer` template tag library. `render_region` and `reverse_app` have been made available as `feincms3`. The `render_plugin` and `render_plugins` tags will be removed completely.
- Changed `feincms3.regions.matches` to the effect that `None` has to be provided explicitly as an allowed subregion if items with no `subregion` attribute should be matched too.
- Removed an use of six which is unnecessary now that we only support Python 3.
- Imported `lru_cache` from the Python library.
- Replaced `concrete_model` calls to determine the concrete subclass of `AppsMixin` with capturing the model instance locally in the `class_prepared` signal handler.
- Removed the now unused `concrete_model` and `iterate_subclasses` utilities.
- Replaced two more occurrences of `.objects` with `._default_manager`.
- Deprecated accessing the backwards compatibility properties `AbstractPage.depth` and `AbstractPage.cte_path`.
- Deprecated `feincms3.apps.AppsMiddleware` in favor of `feincms3.apps.apps_middleware`.

6.1.30 0.28 (2019-02-03)

- **(not yet) BACKWARDS INCOMPATIBLE** Deprecated `TemplatePluginRenderer`'s `regions` method, the `regions_class` attribute and `feincms3.renderer.Regions`. Introduce the more versatile `feincms3.regions.Regions` class instead which also replaces the `feincms3.incubator.subrenderer` functionality and does not suffer from a software design problem where the regions and the renderer classes knew too much about each other. This has been bothering me for a long time already but became impossible to overlook in the subrenderer implementation.
- Updated the Travis CI matrix to cover more versions of Django and Python while reducing the total job count to speed up builds.
- Made the default textarea used for editing the HTML plugin smaller.
- Added documentation for the new `reenter` subrenderer hook.
- Augmented the snippet plugin with a way to specify a template-specific plugin context callable.

6.1.31 0.27 (2019-01-15)

- Fixed the CKEditor plugin script to resize the widget to fit the width of the content editor area.
- Added configuration for easily running prettier and ESLint on the frontend code.
- Dropped Python 2 compatibility, again. The first attempt was made almost 30 months ago.
- Changed the subrenderer to use yielding instead of returning fragments.

6.1.32 0.26 (2018-11-22)

- Removed tree fields when loading applications.
- Stopped mentioning the `AppsMixin` in the reference documentation.
- Fixed a few typos and converted more string quotes in the docs.
- Changed the docs to use allow/deny instead of black/white.
- Changed `feincms3.plugins` do not hide import errors from our own modules anymore (again).
- Added a cloning functionality to copy the values of individual fields and also of the pages' content onto other pages.
- Fixed a problem where `Snippet.__str__` would unexpectedly (for Django) return lazy strings.
- Changed the type of `RedirectMixin.redirect_to_page` to `TreeNodeForeignKey` so that the hierarchy is shown in the dropdown.
- Added more careful detection of chain redirects and improved the error messages a bit.
- Made it clearer that `AbstractPage.position`'s value should probably be greater than zero. Thanks to Hannah Cushman for the contribution!

6.1.33 0.25 (2018-09-07)

- **BACKWARDS INCOMPATIBLE** Removed the imports of plugins into `feincms3.plugins`. Especially with the image plugins it could be non-obvious whether the plugin uses django-imagefield or django-versatileimagefield. Instead, the modules are imported so that classes and functions can be referenced using e.g. `plugins.image.Image` instead of `plugins.Image` as before.
- Moved the documentation from autodoc to a more guide-oriented format.
- Changed `TemplatePluginRenderer.render_plugin_in_context` to raise a specific `PluginNotRegistered` exception upon encountering unregistered plugins instead of a generic `KeyError`.
- Made it possible to pass fixed strings (not callables) to `TemplatePluginRenderer.register_string_renderer`.
- Added an incubator in `feincms3.incubator` for experimental modules with absolutely no compatibility guarantees.
- Changed the `TreeAdmin.move_view` to return a redirect to the admin index page instead of a 404 for missing nodes (as the Django admin's views also do since Django 1.11).
- Fixed an edge case in `apps_urlconf` which would generate a few nonsensical URLs if no language is activated currently.
- Made it an error to add redirects to a page which is already the target of a different redirect. Adding redirects to a page which itself already redirects was already an error.

6.1.34 0.24 (2018-08-25)

- Fixed one use of removed API.
- Fixed a bug where the move form "Save" button wasn't shown with Django 2.1.
- Made overriding the `Regions` type used in `TemplatePluginRenderer` less verbose.
- Modified the documentation to produce several pages. Completed the guide for building your own CMS and added a section about customizing rendering using `Regions` subclasses.

6.1.35 0.23 (2018-07-30)

- Switched the preferred quote to " and started using black to automatically format Python code.

Switched to a new library for recursive common table expressions

django-tree-queries supports more database engines, which means that the PostgreSQL-only days of feincms3 are gone.

Incompatible differences are few:

- The attributes on page objects are named `tree_depth` and `tree_path` now instead of `depth` and `cte_path`. If you're using `WHERE` clauses on your querysets change `depth` to `__tree.tree_depth` (or only `tree_depth`). Properties for backward compatibility have been added to the `AbstractPage` class, but of course those cannot be used in database queries.
- django-tree-queries uses the correct definition of node depth where root nodes have a depth of `0`, not `1`.
- django-tree-queries does not add the CTE by default to all queries, instead, users are expected to call `.with_tree_fields()` themselves if they want to use the CTE attributes. For the time being, the `AbstractPageManager` always returns querysets with tree fields.

6.1.36 0.22 (2018-05-04)

- Fixed a problem in `MoveForm` where invalid move targets would crash because of missing form fields to attach the error to instead of showing the underlying problem.
- Made it possible to override the list of apps processed in `apps_urlconf`.
- Converted the apps middleware into a function, now named `apps_middleware`. The old name `AppsMiddleware` will stay available for some undefined time.
- Made the path clash check less expensive by running less SQL queries.
- Made page saving a bit less expensive by only saving descendants when `is_active` or `path` changed.

6.1.37 0.21 (2018-03-28)

- Added a template tag for `reverse_app`.
- **(At least a bit) BACKWARDS INCOMPATIBLE** Switched the preferred image field from django-versatileimagefield to django-imagefield. The transition should mostly require replacing `versatileimagefield` with `imagefield` in your settings etc., adding the appropriate `IMAGEFIELD_FORMATS` setting and running `./manage.py process_imagefields` once. Switch from `feincms3[all]` to `feincms3[versatileimagefield]` to stay with django-versatileimagefield for the moment.

6.1.38 0.20 (2018-03-21)

- Changed `render_list` and `render_detail` to return `TemplateResponse` instances instead of pre-rendered instances to increase the shortcuts' flexibility.
- Factored the JSON fetching from `oembed_html` into a new `oembed_json` helper.
- Added Django 2.0 to the Travis CI build (nothing had to be changed, 0.19 was already compatible)
- Changed the `TemplatePluginRenderer` to also work when used standalone, not from inside a template.
- Dropped compatibility with Django versions older than 1.11.
- Changed `AppsMixin.clean_fields` to use `_default_manager` instead of `_base_manager` to search for already existing app instances.
- Changed the page move view to suppress the "Save and add another" button with great force.

6.1.39 0.19 (2017-08-17)

The diff for this release is big, but there are almost no changes in functionality.

- Minor documentation edits, added a form builder example app to the documentation.
- Made `reverse_fallback` catch `NoReverseMatch` exceptions only, and fixed a related test which didn't reverse anything at all.
- Switch to tox for building docs, code style checking and local test running.
- Made the `forms.Media` CSS a list, not a set.

6.1.40 0.18 (2017-05-10)

- Slight improvements to `TreeAdmin`'s alignment of box drawing characters.
- Allow overriding the outer namespace name used in `feincms3.apps` by setting the `LANGUAGE_CODES_NAMESPACE` class attribute of the pages class. The default value of `language-codes` has been changed to `apps`. Also, the outer instance namespaces of apps are now of the form `<LANGUAGE_CODES_NAMESPACE>-<language_code>` (example: `apps-en` for english), not only `<language_code>`. This makes namespace collisions less of a concern.

6.1.41 0.17.1 (2017-05-02)

- Minor documentation edits.
- Added the `AncestorFilter` for filtering the admin changelist by ancestor. The default setting is to allow filtering by the first two tree levels.
- Switched from feincms-cleanse to html-sanitizer which allows configuring the allowed tags and attributes using a `HTML_SANITIZERS` setting.

6.1.42 0.16 (2017-04-24)

- Fixed the releasing-via-PyPI configuration.
- Removed strikethrough from our recommended rich text configuration, since feincms-cleanse would remove the tag anyway.
- Made `TemplatePluginRenderer.regions` and the `Regions` class into documented API.
- Made `register_template_renderer`'s `context` argument default to `default_context` instead of `None`, so please stop passing `None` and expecting the default context to work as before.
- Before adding Python 2 compatibility, a few methods and functions had keyword-only arguments. Python 2-compatible keyword-only enforcement has been added back to make it straightforward to transition back to keyword-only arguments later.

6.1.43 0.15 (2017-04-05)

- Dropped the `is_descendant_of` template tag. It was probably never used without `include_self=True`, and this particular use case is better covered by checking whether a given primary key is a member of `page.cte_path`.
- Dropped the `menu` template tag, and with it also the `group_by_tree` filter. Its arguments were interpreted according to the long-gone django-mptt and it promoted bad database querying patterns.
- Dropped the now-empty `feincms3_pages` template tag library.
- Added a default manager implementing `active()` to `AbstractPage`.

6.1.44 0.14 (2017-03-14)

- Removed Django from `install_requires` so that updating feincms3 without updating Django is easier.
- Allowed overriding the Page queryset used in `page_for_app_request` (for example for adding `select_related`).
- Moved validation logic in varous model mixins from `clean()` to `clean_fields(exclude)` to be able to attach errors to individual form fields (if they are available on the given form).
- Added Django 1.11 to the build matrix on Travis CI.
- Fixed an "interesting" bug where the `TreeAdmin` would crash with an `AttributeError` if no query has been run on the model before.

6.1.45 0.13 (2016-11-07)

- Fixed oEmbed read timeouts to not crash but retry after 60 seconds instead.
- Added the `TemplatePluginRenderer.regions` helper and the `{% render_region %}` template tag which support caching of plugins.
- Disallowed empty static paths for pages. `Page.get_absolute_url()` fails with the recommended URL pattern when `path` equals `''`.
- Added flake8 and isort style checking.
- Made the dependency on feincms-cleanse, requests and django-versatileimagefield less strong than before. Plugins depending on those apps simply will not be available in the `feincms3.plugins` namespace, but you have to be careful yourself to not import the actual modules yourself.

- Added Django, django-content-editor and django-cte-forest to `install_requires` so that they are automatically installed, and added an extra with dependencies for all included plugins, so if you want that simply install `feincms3[all]`.

6.1.46 0.12 (2016-10-23)

- Made `reverse_any` mention all viewnames in the `NoReverseMatch` exception instead of bubbling the last viewname's exception.
- Added a `RedirectMixin` to `feincms3.mixins` for redirecting pages to other pages or arbitrary URLs.
- Added a footgun plugin (raw HTML code).
- Reinstate Python 2 compatibility because Python 2 still seems to be in wide use.

6.1.47 0.11 (2016-09-19)

- Changed the implementation of the `is_descendant_of` template tag to not depend on django-mptt's API anymore, and removed the compatibility shims from `AbstractPage`.
- Made the documentation build again and added some documentation for the new `feincms3.admin` module.
- Made `TreeAdmin.move_view` run transactions on the correct database in multi-DB setups.
- Removed the unused `NoCommitException` class.
- Fixed a crash in the `MoveForm` validation.
- Made `AppsMiddleware` work with Django's `MIDDLEWARE` setting.
- Made the `{% menu %}` template tag not depend on a `page` variable in context.

6.1.48 0.10 (2016-09-13)

- **BACKWARDS INCOMPATIBLE** Switched from django-mptt to django-cte-forest which means that feincms3 is for the moment PostgreSQL-only. By switching we completely avoid the MPTT attribute corruption which plagued projects for years. The *lft* attribute is directly reusable as *position*, and should be renamed in a migration instead of created from scratch to avoid losing the ordering of nodes within a branch.
- Added a `feincms3.admin.TreeAdmin` which shows the tree hierarchy and has facilities for moving nodes around.
- Avoided a deprecation warning on Django 1.10 regarding `django.core.urlresolvers`.
- Started rolling releases using Travis CI's PyPI deployment provider.
- Made `{% is_descendant_of %}` return `False` if either of the variables passed is no page instance instead of crashing.

6.1.49 0.9 (2016-08-17)

- Dropped compatibility with Python 2.
- Fixed `AbstractPage.save()` to actually detect page moves correctly again. Calling `save()` in a transaction was a bad idea because it messed with MPTT's bookkeeping information. Depending on the transaction isolation level going back to a clean slate *after* `clean()` proved much harder than expected.

6.1.50 0.8 (2016-08-05)

- Added `feincms3.apps.reverse_fallback` to streamline reversing with fallback values in case of crashes.
- The default template renderer context (`TemplatePluginRenderer.register_template_renderer`) contains now the plugin instance as `plugin` instead of nothing.
- Make django-mptt-nomagic a required dependency, by depending on the fact that nomagic always calls `Page.save()` (django-mptt does not do that when nodes are moved using `TreeManager.node_move`, which is used in the draggable mptt admin interface. Use a `node_moved` signal listener which calls `save()` if the `node_moved` call includes a `position` keyword argument if you can't switch to django-mptt-nomagic for some reason.

6.1.51 0.7 (2016-07-21)

- Removed all dependencies from `install_requires` to make it easier to replace individual items.
- Enabled the use of `i18n_patterns` in `ROOT_URLCONF` by importing and adding the urlpatterns contained instead of `include()`-ing the module in `apps_urlconf`.
- Modified the cleansing configuration to allow empty <a> tags (mostly useful for internal anchors).
- Fixed crash when adding a page with a path that exists already (when not using a statich path).

6.1.52 0.6 (2016-07-11)

- Updated the translation files.
- Fixed crashes when path of pages would not be unique when moving subtrees.

6.1.53 0.5 (2016-07-07)

- Fixed a crash where apps without `required_fields` could not be saved.
- Added a template snippet based renderer for plugins.
- Prevented adding the exact same application (that is, the same `app_instance_namespace`) more than once.

6.1.54 0.4 (2016-07-04)

- Made application instances (`feincms3.apps`) more flexible by allowing programmatically generated instance namespace specifiers.

6.1.55 0.3 (2016-07-02)

- Lots of work on the documentation.
- Moved all signal receivers into their classes as staticmethods.
- Fixed a crash on an attempted save of an `External` plugin instance with an empty URL.
- Added an incomplete testsuite, and add the Travis CI badge to the README.
- Removed the requirement of passing a context to `render_list` and `render_detail`.

6.1.56 0.2 (2016-06-28)

- The external plugin admin form now checks whether the URL can be embedded using OEmbed or not.
- Added the `plugin_ckeditor.js` file required for the rich text editor.
- Added a `SnippetInline` for consistency.
- Ensured that choice fields have a `get_*_display` method by setting dummy choices in advance (menus, snippets and templates).
- Added automatically built documentation on readthedocs.io.

6.1.57 0.1 (2016-06-25)

- Plugins (apps, external, richtext, snippet and versatileimage) for use with django-content-editor.
- HTML editing and cleansing using django-ckeditor and feincms-cleanse.
- Shortcuts (`render_list` and `render_detail` – the most useful parts of Django's class based generic views)
- An abstract page base model building on django-mptt with mixins for handling templates, menus and language codes.
- Template tags for fetching and grouping menu entries inside templates.
- A german translation.

6.2 Contributing

This isn't a Jazzband project, but by contributing you agree to abide to the same Contributor Code of Conduct as if it was one.

6.2.1 Bug reports and feature requests

You can report bugs and request features in the bug tracker.

6.2.2 Code

The code is available on GitHub.

To work on the code I strongly recommend installing tox. I use tox as a glorified virtualenv-builder and task runner for local development.

Available tasks are:

- `tox -e style`: Reformats the code using black and runs flake8.
- `tox -e docs`: Builds the HTML docs into `build/docs/html/`
- `tox -e py??-dj?'`: Runs tests using combinations of Python and Django. See `tox -l` for all available combinations.

Both testing tasks also generate HTML-based code coverage output into the `htmlcov/` folder.

6.2.3 Style

Python code for the feincms3 project may be automatically formatted and checked using `tox -e style`. The coding style is also checked when building pull requests using Github actions.

6.2.4 Patches and translations

Please submit pull requests!

I am not using a centralized tool for translations right now, I'll happily accept them as a patch too.

6.2.5 Mailing list

If you wish to discuss a topic, please open an issue on Github. Alternatively, the django-feincms Google Group may also be used for discussing this project.

CHAPTER

SEVEN

RELATED PROJECTS

- feincms3-example: Example project demonstrating some of feincms3's capabilities.
- feincms3-sites: Multisite support for feincms3. Allows running a feincms3 site on several domains with separate page trees.
- feincms3-downloads: A downloads plugin which also supports thumbnailing e.g. PDFs using ImageMagick.
- feincms3-meta: Helpers and feincms3 mixins for making Open Graph tags and meta tags less annoying.
- feincms3-forms: A form builder using django-content-editor under the hood.
- django-cabinet: A media library for Django which works well with feincms3 and follows the same software design guidelines.
- django-content-editor: The admin interface for editing structured heterogenous content.
- django-imagefield: An image field with in-depth image file validation and thumbnailing support which does not depend on a cache to be and stay fast.
- django-sitemaps: Sitemaps generation using a real XML library and support for alternates.
- django-tree-queries: The library feincms3's pages use for querying tree-shaped data.
- html-sanitizer: Allowlist-based HTML sanitizer used for feincms3' rich text plugin.
- FeinCMS: First version.

PYTHON MODULE INDEX

f

INDEX

G

H

I

L

M

V